O MÉTODO DE GERAÇÃO DE CÓDIGO

DOS COMPILADORES LPS

Ivan Luís Gonçalves de Oliveira Lima

TESE SUBMETIDA AO CORPO DOCENTE DA COORDENAÇÃO DOS PROGRAMAS DE PÓS-GRADUAÇÃO EM ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS (M. Sc.).

Aprovada por:

Estevam Gilberto De Simone
Presidente

José Lucas Mourão Rangel Netto

Eduardo Lessa Peixoto de Azevedo

RIO DE JANEIRO, RJ - BRASIL
FEVEREIRO DE 1982

OLIVEIRA LIMA, IVAN LUÍS GONÇALVES DE

O Método de Geração de Código dos Compiladores LPS (Rio de Janeiro) 1982.

VII,137p. 29,7cm (COPPE-UFRJ, M. Sc., Engenharia de Sistemas, 1982).

Tese - Universidade Federal do Rio de Janeiro, Programa de Engenharia de Sistemas da Coordenação dos Programas de Pós-Graduação em Engenharia.

1.Compiladores I.COPPE/UFRJ II.Título(série).

RESUMO:

A tarefa de geração de código por um compilador é dividida em três fases, que podem ser executadas paralela ou consecutivamente. Para cada uma delas são descritas técnicas auxiliares na sua especificação e implementação, bem como a aplicação destas técnicas a compiladores em um passo com análise sintática ascendente, e em especial aos compiladores LPS para as máquinas COBRA-300 e COBRA-500.

A primeira destas fases é a tradução do programa fonte em uma sequência de chamadas a rotinas semânticas. Mediante a introdução de um segundo conjunto de símbolos terminais - que representam estas rotinas - em uma gramática livre-de-contexto, são definidas gramáticas de tradução livres-de-contexto, que permitem a especificação de um tradutor comandado pelo analisador sintático.

A segunda fase consiste na geração de código intermediário pelas rotinas semânticas. Para esta fase, verificamos a necessidade de variáveis de inter-comunicação das rotinas semânticas, que denominamos atributos semânticos. Com as gramáticas de tradução livres-de-contexto com atributos, que são uma extensão das gramáticas de tradução livres-de-contexto, pode ser descrita a maneira pela qual o analisador sintático dirigirá também o armazenamento e propagação dos atributos semânticos.

A geração de código se completa com a tradução do código intermediário em instruções do processador. No caso das operações binárias esta tradução não é imediata. Definimos então um esquema para tradução de operações binárias, que permite transformar a escolha da melhor sequência de instruções para realizar uma operação binária num problema de determinação do caminho mínimo entre dois nós de um digrafo com custos nas arestas.

O método ora descrito propicia a obtenção de compiladores facilmente portáteis para um novo processador objeto, onde o principal esforço será de redefinição do esquema para tradução de operações binárias, tendo em vista o novo processador.

ABSTRACT:

Code generation is divided into three phases, which can be executed either sequentially or in parallel. We describe some techniques that can be helpful in specifying and implementing each one of them. We examine as well the application of those techniques in one-pass compilers with bottom-up parsers, specially COBRA-300 and COBRA-500 LPS compilers.

In the first phase, the source program is translated into a sequence of semantic routine calls. By adding a second set of terminal symbols - which stand for the semantic routines - to a context-free grammar, we define context-free translation grammars, used to specify a parser-driven translator.

In the second phase, the semantic routines are executed, generating intermediate code. To do so, there must exist intercommunication variables, called semantic attributes. Extending context-free translation grammars to include also a set of attributes, the storage and propagation of semantic attributes can also be parser-driven.

Code generation will be completed by the translation of intermediate code into processor instructions. Such translation process is not straightforward for binary operations. We then define a binary operations translation scheme, which transforms the problem of choosing the best sequence of processor instructions to perform an intermediate binary operation into the problem of finding the shortest path between two nodes of a weighted digraph.

Compilers generated in the way just described are easily portable to a new target processor, where the main effort will be the redefinition of the binary operations translation scheme.

ÍNDICE

AGRADECIMENTOS

Entre todos aqueles que direta ou indiretamente contribuíram para a realização deste trabalho, gostaria de agradecer, antes de mais nada, ao Estevam, pelos ensinamentos valiosos e amável orientação. Agradeço também a meus pais, por razões óbvias; a meus avós, que me acompanharam no mês decisivo de estudo e primeira redação, em Petrópolis; e à Lila, pelo apoio e compreensão irrestritos durante o longo período de redação final. Dentre os colegas e amigos da COBRA, gostaria de destacar Andréa Moraes, Marco Antônio Tiso, Tadeu Filgueiras de Souza, Luiz Fernando Seibel e Lys Caldas, companheiros nos projetos LPM e LPS; Rosana Lanzelotte, que além de amiga e colega especial, ajudou com disposição sempre que necessário; Jorge da Silveira, pela imensa boa-vontade com que realizou a digitação deste trabalho no SPP; Eduardo Lessa, que vem sempre apoiando e garantindo a melhor formação dos funcionários da DDS; e finalmente o Luiz Carlos Monte, que além de meu iniciador na ciência da computação, pelo menos em termos de idéias deve ser considerado co-autor deste trabalho.

I - Introdução

O processo de compilação pode ser dividido, a grosso modo, nas fases de análise léxica, análise sintática e geração de código. Enquanto que, para as duas primeiras, extensamente estudadas, já se dispõe de descrições adequadas para o seu funcionamento esperado, possibilitando assim a automatização completa da implementação de analisadores léxicos e sintáticos para a maior parte das linguagens de programação correntes, tal não acontece com a geração de código. Para esta fase da compilação, entretanto, diversas ferramentas auxiliares foram definidas, como os esquemas da tradução dirigidos pela sintaxe, ou SDTS (AHO, 1), procurando melhor estruturar a tarefa de geração de código.

Este trabalho procura contribuir neste sentido. Trata-se de uma organização formal dos conhecimentos adquiridos no projeto e implementação do sistema de compiladores LPS para os computadores COBRA-300 e COBRA-500. Assim, sem a pretensão de definir um programa gerador de geradores de código, procuramos decompor esta fase da compilação em algumas etapas, que podem ser executadas simultânea ou consecutivamente, e estabelecer formas de descrição do funcionamento de cada uma delas, bem como algoritmos que facilitem sua implementação. Foi dada também especial atenção ao problema de portabilidade do compilador enquanto tradutor para mais de uma linguagem objeto, como os conjuntos de instruções dos processadores do COBRA-300 e COBRA-500. Para tanto, procurou-se reduzir ao mínimo a parte do compilador dependente do processador objeto, bem como estabelecer diretrizes que tornassem a implementação desta parte o mais automática possível.

I.1 - A Linguagem LPS

A linguagem LPS (Linguagem para Programação de Sistemas) foi projetada na COBRA, para ser um ferramenta no desenvolvimento de software para seus produtos. Nela foi escrito todo o software básico dos produtos da linha COBRA-300 (TD-100, TD-200, COBRA-300, COBRA-305), e boa parte do software básico da linha COBRA-500. No caso deste último, a LPS facilitou também a trans-

ferência de programas originalmente escritos para a linha 300. Atualmente, programas podem ser escritos em LPS e utilizados em ambas as máquinas, com modificação apenas das partes dependentes do computador e sistema operacional hospedeiros.

Baseada no ALGOL-60, a LPS é uma linguagem hîbrida que inclui declarações e comandos de alto nîvel semelhantes aos do ALGOL, mas admite também a escrita de trechos do programa diretamente na linguagem simbólica (assembly) do processador. Naturalmente, esta facilidade não pode ser utilizada por programas que se pretenda portáteis entre os dois processadores objeto, mas é de fundamental importância na escrita de programas fortemente vinculados ao processsador e que precisem de acesso irrestrito a todas as partes da máquina, tais como sistemas operacionais.

Todos os identificadores de um programa LPS devem ser declarados e, sendo o programa estruturado em blocos, são válidas as regras usuais para o alcance de uma declaração. Podem ser declarados em LPS identificadores de constantes, variáveis, rótulos, chaves e procedimentos. São dois os tipos de dados admitidos: inteiros de 8 bits, definidos como tipo "byte", e inteiros de 16 bits, do tipo "word". Estes dados podem ser tratados como números com ou sem sinal, em função do contexto, mas existe uma opção principal ("default") controlado pelo registro inicial do programa fonte. A LPS permite ainda a subdivisão de um programa em módulos compilados em separado, e posteriormente ligados por um programa relocador. A comunicação entre estes módulos se faz por intermédio de variáveis e procedimentos declarados globais em um módulo e externos em outros. A descrição completa de linguagem LPS pode ser encontrada em (10).

I.2 - O Sistema de Compiladores LPS

O Sistema de compiladores LPS foi elaborado por ocasião da feitura do compilador LPS para o COBRA-300 na própria linguagem, da qual já se dispunha de um compilador no computador HP-21MX. Consta de quatro compiladores, com as máquinas COBRA-300 e COBRA-500 como hospedeiro ou objeto. Destes, já estão em uso o

compilador LPS executável no COBRA-300 gerando código para seu próprio processador (LPS/300), o compilador LPS executável no COBRA-500 gerando código para seu próprio processador (LPS/500) e o compilador LPS executável no COBRA-300 gerando código para o COBRA-500 (LPS500). O método de análise sintática utilizado é o de matrizes de transição (GRIES, 6).

Os compiladores LPS admitem como entrada três tipos de arquivos: um arquivo fonte principal obrigatório e, opcionalmente, um arquivo de alterações sobre o fonte principal e vários arquivos com registros a serem incluídos na compilação. Fornecem como saída um arquivo de relatório (listagem), um arquivo com o código objeto gerado e o arquivo resultante da efetivação dos comandos do arquivo de alteração sobre o arquivo fonte principal. Todas as três saídas são opcionais. O código objeto compõe-se de uma lista de identificadores globais e externos, e uma sequência de campos de relocação absoluta, relocáveis em dados ou programa ou correspondentes a referências externas. Nestes últimos são dados o número de ordem da referência externa e um deslocamento com relação ao endereço de resolução desta referência. Uma lista de informações sobre a relocação define o tipo de relocação de cada campo do texto objeto. No âmbito deste trabalho, porém, consideraremos como código objeto a saída do gerador de código, que inclui ainda palavras contendo números a serem associados a endereços de programa, originados por referências à frente, e uma lista de definições destes números.

Nos compiladores LPS, análise léxica, sintática e geração de código são realizadas num mesmo passo, mas o código objeto é percorrido uma segunda vez, quando os números das referências à frente são substituídos pelos endereços correspondentes, obtidos da lista de definições. A saída deste segundo passo é um módulo relocável, e vários destes módulos podem ser ligados pelos programas relocadores do COBRA-300 e COBRA-500, que também resolvem as referências externas. Maiores informações sobre os compiladores LPS podem ser obtidas em (10) e em (11).

I.3 - Organização do Texto

Este texto compõe-se de seções teóricas e práticas. Nas primeiras procura-se descrever métodos para elaboração das fases em que foi dividida a geração de código. Nas seções práticas são ilustradas todas as fases da geração de código, sempre por intermédio da aplicação dos procedimentos teóricos aos compiladores LPS, para as máquinas COBRA. Além deste capítulo introdutório, no qual é estabelecido o propósito e feita uma descrição sucinta do sistema LPS para ambientação da tese, e do capítulo de conclusão, três outros capítulos tratam de igual número de etapas da geração de código.

A primeira etapa consiste na tradução do programa fonte em uma sequência de chamadas a rotinas semânticas. Para tanto, são definidas as gramáticas de tradução livres de contexto, e estudadas a seguir sua aplicação a analisadores sintáticos ascendentes, em especial por matrizes de transição, bem como sua utilização pelos compiladores LPS. Esta é a constituição da primeira seção do segundo capítulo da tese.

A segunda etapa da geração de código é a execução das rotinas semânticas, durante a qual é emitido um código intermediário e feito o cálculo de atributos semânticos. Embora a definição do funcionamento das rotinas semânticas seja feita em linguagem corrente ou em forma similar a alguma linguagem de programação, o armazenamento e propagação dos atributos semânticos pode ser descrito em conjunto com a sintaxe, e controlado pelo analisador sintático. Isto é feito com o auxílio das gramáticas de tradução livres-de-contexto com atributos, definidas na segunda seção do segundo capítulo, onde é também examinada sua aplicação com analisadores sintáticos ascendentes e aos compiladores LPS. A terceira seção do segundo capítulo descreve um código intermediário para LPS.

O terceiro capítulo exemplifica a fase de execução das rotinas semânticas, mostrando o cálculo de atributos e a geração de código intermediário pelas rotinas semânticas LPS. Com isto procuramos verificar que rotinas são necessárias para a tradução das construções normalmente encontradas nas linguagens de pro-

mação usuais, bem como estudar seu funcionamento e inter-relacionamento.

A terceira fase da geração de código é a tradução do código intermediário em instruções do processador objeto. Esta transformação é quase imediata para boa parte das instruções intermediárias, se não considerarmos o contexto em que se encontram. No caso das operações binárias, porém, existe muitas vezes grande diversidade de situações em que cada operando pode se encontrar, e modificações na localização de um operando devem ser feitas levando-se em conta o outro operando, até que se obtenha uma combinação para a qual a operação binária em questão possa ser realizada. Desta terceira fase de geração de código trata o quarto capítulo deste trabalho, onde também é vista a aplicação do esquema acima descrito ao compilador LPS gerando código para o processador INTEL-8080 do COBRA-300 e para o processador COBRA-500.

Não obstante forneça as diretrizes básicas da geração de código pelos compiladores LPS, este trabalho não pode ser utilizado como manual de lógica, pois embora as idéias gerais do método exposto tenham sido estabelecidas por ocasião do projeto deste sistema de compiladores, sua formalização foi posterior à implementação dos compiladores, e as aplicações aqui descritas referem-se ao método formalizado.

II - Método de Geração de Código Intermediário

A definição de um código intermediário é um artifício de grande valia no projeto de um compilador. Permite a separação do problema de geração de código em duas fases: a primeira, ligada à análise sintática, produz o código intermediário, que já traz definida a estrutura geral do código objeto, enquanto que a segunda fase se preocupará com a escolha de uma sequência de instruções do processador-objeto que traduza o programa de forma eficiente, usando adequadamente os registradores do processador. A maior facilidade com que podem ser efetuadas otimizações no código intermediário, em relação ao código final, também é um atrativo para a escolha deste processo, mas sua maior vantagem é, talvez, a maior portabilidade do produto obtido, pois apenas a fase de tradução do código intermediário em instruções do processador é dependente da máquina-objeto. Além disso, para um mesmo código intermediário, suficientemente geral, podemos ter mais de uma linguagem fonte, ou novas versões da mesma linguagem sendo traduzidas.

O principal ônus deste procedimento é o aumento do tempo de compilação, provocado pela criação de um arquivo de código intermediário. Mesmo assim, se o compilador dispõe de pouca memória e precisa ser segmentado, a separação destas duas etapas em dois segmentos de sobreposição será provavelmente mais vantajosa do que a separação em segmentos que se alternem frequentemente na memória.

A execução das duas etapas anteriores, por outro lado, pode ser simultânea, dispensando a criação do arquivo intermediário e diminuindo o tempo de compilação. Neste caso, porém, a otimização sobre o código intermediário fica dificultada, bem como torna-se ineficiente a separação das duas etapas em segmentos de sobreposição. Neste esquema, o primeiro módulo gera da mesma forma um código intermediário, que no entanto é imediatamente traduzido para o código da máquina objeto por um segundo módulo que, mantendo as características da segunda fase descrita a princípio, é executado a cada intrução intermediária emitida.

A geração de código intermediário, por sua vez, será também dividida em duas fases, sendo a primeira de tradução do programa

em uma sequência de chamadas a rotinas semânticas e a segunda de execução destas rotinas, quando serão emitidas as instruções intermediárias.

II.1 - Tradução do Programa em Rotinas Semânticas.

A escolha de um método conveniente para se proceder à esta tradução permitirá que a vinculemos totalmente à sintaxe da linguagem, de tal modo que na prática as chamadas às rotinas semânticas farão parte do corpo do analisador e serão efetuadas à medida que é feito o reconhecimento do programa.

Apresentamos a seguir uma solução formal do problema com a definição de uma gramática de tradução livre-de-contexto (GTLC), onde a tradução é especificada nas próprias regras sintáticas. Analisamos em sequência os cuidados que se deve tomar quando o analisador sintático for ascendente e finalmemte descrevemos a aplicação destas considerações aos compiladores LPS.

Para a definição de GTLC baseamo-nos nas gramáticas de tradução livres-de-contexto estendidas (Lewi,7), cujo emprego, porém, foi evitado, por não se adaptar ao método de análise sintática por matrizes de transição.

II.1.1 - Gramáticas de Tradução Livres-de-Contexto

Uma gramática de tradução livre-de-contexto pode ser entendida simplesmente com uma gramática livre-de-contexto acrescida de um outro conjunto de símbolos terminais, cujos elementos podem aparecer do lado direito das produções e que serão os símbolos de saída da tradução. A definição é dada a seguir.

Definição: Uma gramática de tradução livre-de-contexto (GTLC) é uma 5-tupla G = (Ve,Vs,Vn,S,P) onde

1 - Ve é um conjunto finito de símbolos de entrada, ou vocabulário de entrada.

2 - Vs é um conjunto finito de símbolos de saída, ou vocabulário de saída, tal que Ve $\cap$ Vs = Φ.

3 - Vn é um conjunto de símbolos não-terminais, tal que Vn $\cap$ Ve = Φ e Vn $\cap$ Vs = Φ .

4 - S é o símbolo inicial de G, S $\in$ Vn.

5 - P é um conjunto finito de regras da forma A ::= α , sendo A $\in$ Vn e α $\in$ (Ve **u** Vs u Vn)*. Os elementos de P são denominados produções e a relação A ::= α lê-se A produz α.

Definição: A relação ==> (deriva diretamente) é uma relação sobre
(Ve u Vs u Vn)+ X (Ve u Vs u Vn)*. Dizemos que $\alpha A \beta$ ==> $\alpha \gamma \beta$ se e somente se (A ::= γ) $\in$ P.

Definição: A relação $\overset{*}{\Longrightarrow}$ (deriva) é uma relação em
(Ve u Vs u Vn)+ X (Ve u Vs u Vn)*. Dizemos que α ==> β se existem
$\alpha 0, \alpha 1, \ldots, \alpha n \in$ (Ve u Vs u Vn)*, n >= 0 tais que $\alpha 0$ ==> $\alpha 1$ ==> ... ==> αn, $\alpha 0 = \alpha$ e $\alpha n = \beta$.

Definição: Forma sentencial de G é um elemento $\delta \in$ (Ve u Vs u Vn)* tal que S $\overset{*}{\Longrightarrow} \delta$.

Definição: Derivação de uma forma sentencial δ é uma sequência $\alpha 0, \alpha 1, \ldots, \alpha n$, n >= 0, de elementos de (Ve **u** Vs **u** Vn)* tal que $\alpha 0$ ==> $\alpha 1$ ==> ...==> αn, $\alpha 0$ = S e $\alpha n = \delta$.

Definição: Sentença de G é uma forma sentencial ω de G tal que $\omega \in$ (Ve u Vs)*. A linguagem L(G) definida por G é o conjunto de todas as sentenças de G.

Definição: Parte de entrada de uma sentença de G é a cadeia obtida ao eliminarmos da sentença todos os símbolos de Vs. Parte de saída de uma sentença de G é a cadeia obtida ao eliminarmos desta todos os símbolos de Ve.

Definição: Tradução em G é o conjunto de todos os pares (parte de entrada, parte da saída) das sentenças de G, denotado T(G). Isto é, $T(G) = \{(\alpha,\beta) \mid \exists\ \delta \in (Ve \cup Vs)^*, S \overset{*}{\Rightarrow} \delta, \alpha$ é a parte de entrada de δ e β é a parte de saída de $\delta\}$.

Definição: Gramática de entrada de G é G1 = (Ve, Vn, S, P1) onde P1 é obtido eliminando-se todos os símbolos de saída das produções de P. A linguagem L(G1) definida por G1 é o conjunto de todas as cadeias $\alpha \in (Ve)^*$ para as quais existe uma tradução β em G, ou seja
$L(G1) = \{ \alpha \in (Ve)^* \mid \exists\ \beta \in (Vs)^*$ e $(\alpha,\beta) \in T(G)\}$. **Gramática** de saída G2 e linguagem de saída L(G2) são definidas analogamente.

Definição: Uma árvore de derivação de uma formal sentencial de G é uma árvore com nós rotulados construída da seguinte forma:
a raiz é rotulada S:
a cada passo da derivação de onde foi usada a produção
A ::= B1 B2 ... Bn, Bi $\in$ (Ve $\cup$ Vs $\cup$ Vn) para $1 <= i <= n$, crie n filhos para o nó rotulado A e rotule-os da esquerda para a direita B1, B2, ..., Bn.

Definição: Diremos que um nó x de uma árvore de derivação precede um nó y da mesma árvore se x é visitado antes de y ao percorrermos esta árvore em pré-ordem (raiz, sub-árvore esquerda, sub-árvore direita).

Notação: Adotaremos as seguintes regras para os símbolos de uma GTLC, ao longo de todo este trabalho:
1 - Os símbolos não-terminais serão representados por cadeias de letras maiúsculas, admitindo-se também dígitos numéricos e o caráter "." (ponto), estes últimos exceto na primeira posição.
2 - Os símbolos terminais serão representados por cadeias de letras minúsculas e demais caracteres, excetuando-se as letras maiúsculas. Além disso, os símbolos terminais de saída serão distinguidos dos de entrada através de grifo.

Com relação às produções de uma GTLC, adotaremos ainda a seguinte regra:

3 - A notação A::= α1, A::=α2, ..., A::=αn, onde (A::=αi) ε P para 1<=i<=n, e a notação A::=α1 | α2 | ... | αn são equivalentes.

O exemplo a seguir ilustra a utilização das GTLC para definir uma tradução e obter a parte de saída dada a parte de entrada de uma sentença.

Exemplo 1 : Tradução de expressões aritméticas infixas para notação posfixa.
Ga = (Ve, Vs, Vn, E, P)

Ve = a, b, ..., z, 0, 1, ..., 9, +, -, *, /, (,)

Vs = a, b, ..., z, 0, 1, ..., 9, +, -, *, /, _

Vn = E, T, F, IDENTIFICADOR, LETRA, DIGITO, NUMERO

E é o símbolo inicial

As regras de P são:

```
E ::= T
    | E + T+
    | E - T-
T ::= F
    | T * F*
    | T / F/

F ::= IDENTIFICADOR _
    | NÚMERO _
    | (E)

IDENTIFICADOR ::= LETRA
                | IDENTIFICADOR LETRA
                | IDENTIFICADOR DÍGITO
```

```
NÚMERO ::= DÍGITO
         | NÚMERO DÍGITO
LETRA ::= a a
        | b b
        .
        .
        .
        | z z

DÍGITO ::= 0 0
         | 1 1
         | .
         | .
         | .
         | 9 9
```

Consideremos a expressão ab+cd*2. Sua árvore de derivação na gramática de entrada de Ga está na figura 1.

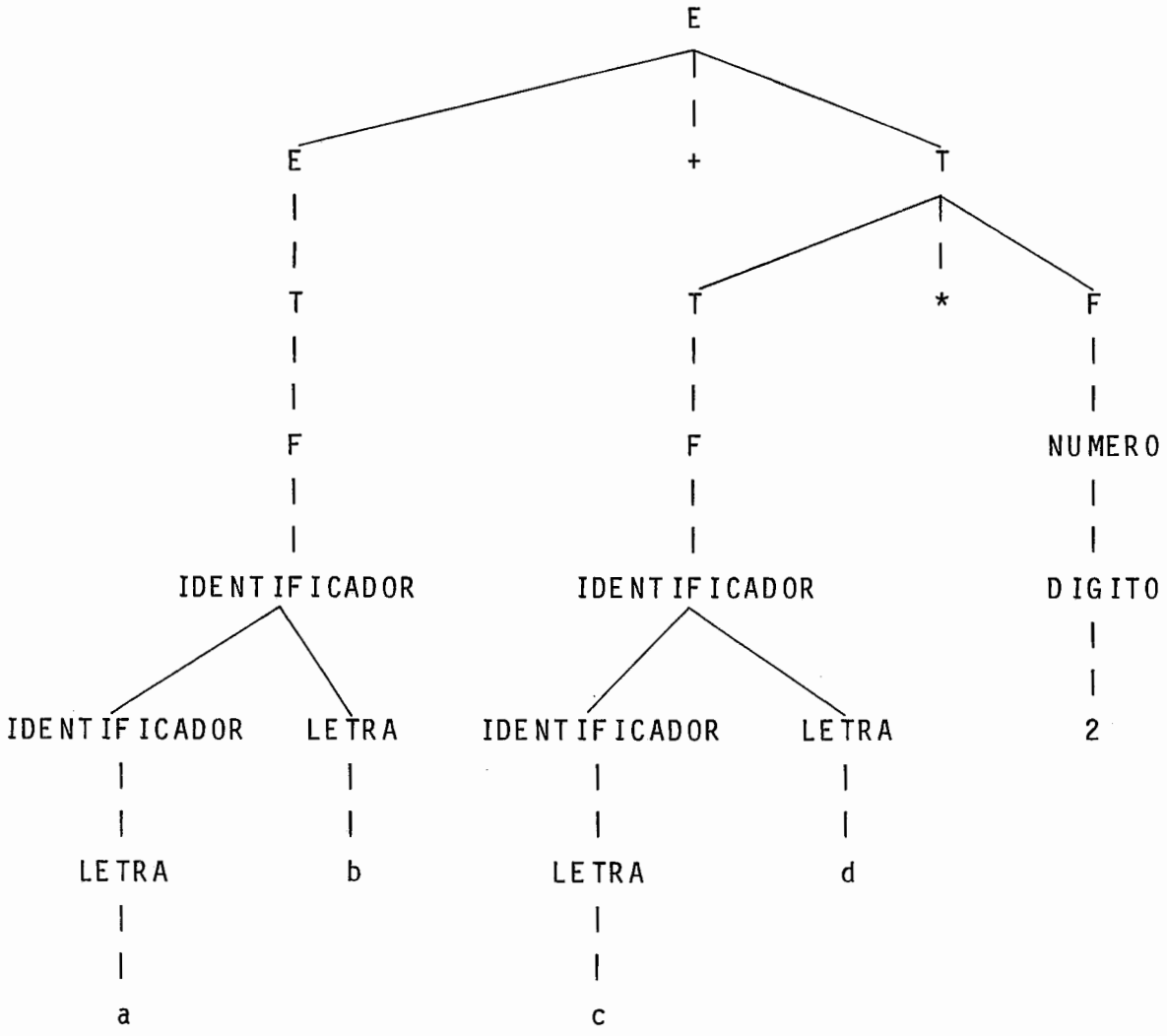


Figura 1 - Árvore de Derivação da Gramática de Entrada

Se repetirmos em Ga a derivação feita na gramática de entrada, poderemos construir a árvore da figura 2.

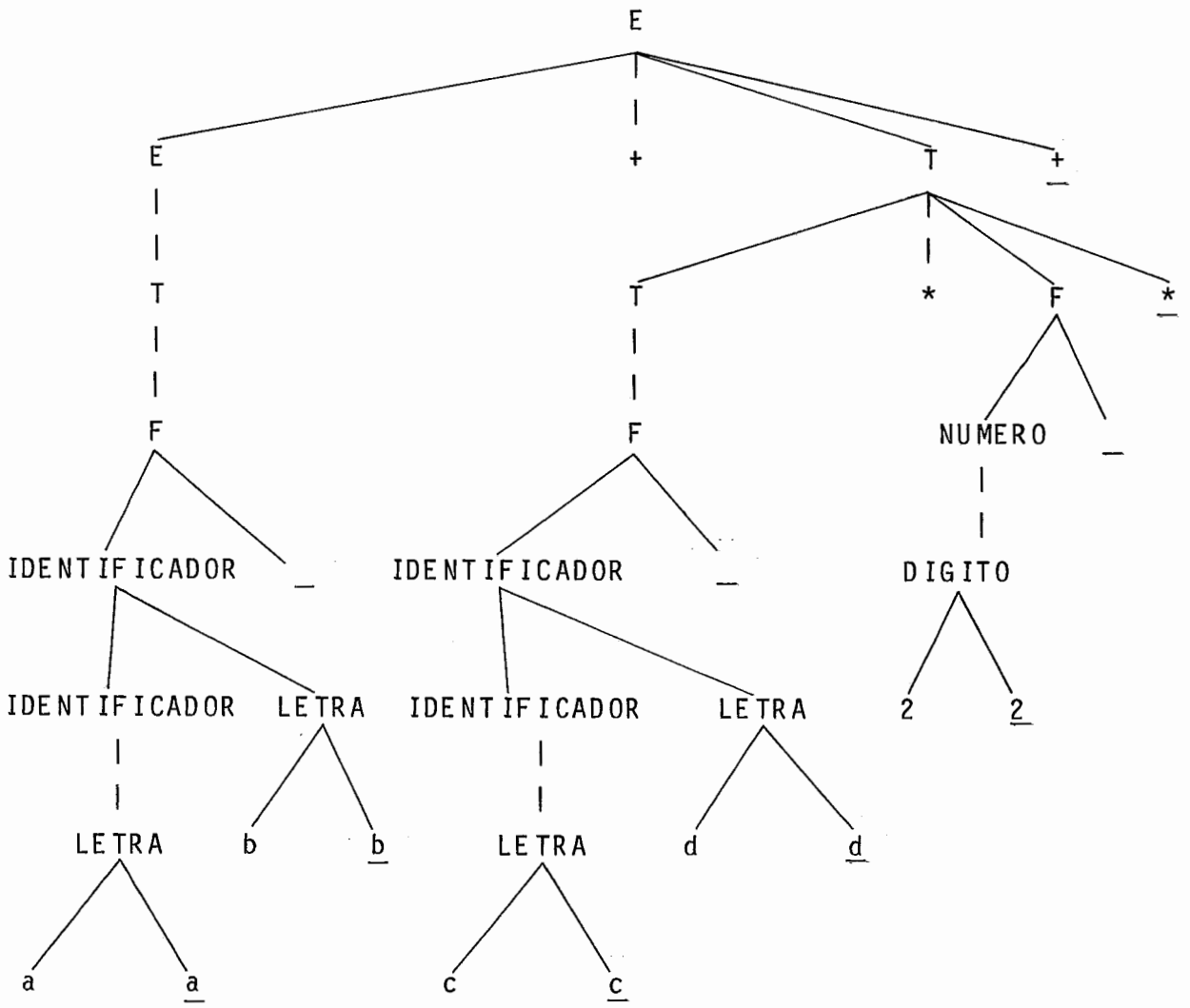


Figura 2 - Árvore de Derivação na GTLC

Esta derivação define a forma sentencial

a a b b _ + c c d d _ * 2 2 _ _ * + ,

cujas partes de entrada e saída são respectivamente

ab+cd*2 e

a b _ c d _ 2 _ * + (que equivale a ab cd 2 *+),

o que significa que a tradução em Ga da expressão ab+cd*2 é a

expressão a b _ c d _ 2 _ * +.

O processo de tradução é, portanto, construir a derivação de uma sentença cuja parte de entrada seja a cadeia a traduzir. A parte de saída desta sentença será a tradução desejada.

II.1.2 - Gramáticas de Tradução Livres-de-Contexto com Análise Sintática Ascendente

No que foi exposto anteriormente procuramos mostrar de que maneira uma GTLC pode ser usada para definir um tradutor, e como podemos obter a tradução a partir da árvore de derivação da sentença de entrada. Este não é, no entanto, um procedimento prático, uma vez que durante a análise sintática esta árvore nunca chega a ser totalmente construída. Os analisadores ascendentes, por exemplo, limitam-se a usar uma pilha para armazenar as informações sobre o que já foi analisado necessárias ao prosseguimento de sua tarefa. Diante disso, torna-se necessário que a tradução seja gerada à medida que a entrada é percorrida, isto é, que a tradução do que foi condensado nas informações da pilha sintática já tenha sido produzida.

O procedimento adotado é o de emitir a parte de saída de cada produção usada na análise sintática da entrada. Ora, a análise sintática ascendente fornece a sequência de produções usadas na redução direita da sentença ao símbolo inicial. Portanto, se pretendemos emitir a sentença de saída à medida que formos progredindo na análise da entrada é necessário que a GTLC satisfaça determinadas condições para que a sentença traduzida seja correta. Vamos inicialmente exemplificar o problema e a seguir determinar restrições sobre as produções das GTLCs que nos forneçam uma subclasse de GTLCs onde a sentença de saída, emitida segundo a ordem de reduções da sentença de entrada, será sempre correta.

Exemplo 2: Gramática de tradução do comando "if-then" em rotinas semânticas.

Gif = (Ve, Vs, Vn, C.IF, P)

Ve = { id, :=, if, then }

Vs = { id, :=, if, then }

Vn = { C.IF, C, E, V }

As regras de P são:

(1) C.IF ::= if E then if C then

(2) C ::= C.IF
(3) | V := E :=

(4) E::= id id

(5) V::= id id

Vejamos as reduções da sentença de entrada "if id then id:=id":

if id1 then id2:=id3 <== if E then id2:=id3 <== if E then V:=id3 <== if E then V:=E <== if E then C <== C.IF

A sentença de saída obtida pela emissão dos símbolos de saída a cada redução é
id1 id2 id3 := if then

Entretanto se fizermos a derivação mais à esquerda correspondente em Gif obteremos a sequência

id1 if id2 id3 := then

que está na ordem natural para geração de código. Veremos agora como solucionar este problema, dentro de nossa GTLC, sem cons-

truir e percorrer a árvore. Isto pode ser feito de diversas formas, através da modificação da gramática original. Uma maneira de o fazermos é criar um ou mais não terminais artificiais que produzam apenas ϕ (vazio) e usar esta redução para emitir a saída desejada. Em nosso caso, substituiríamos a regra (1) por

C.IF ::= if E then M C then
M ::= ϕ if

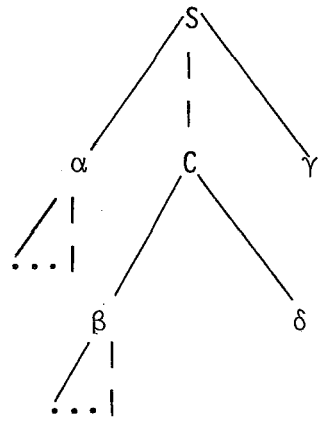
Este processo só é possível se o analisador sintático, como por exemplo os analisadores LR, aceita gramáticas com produções de lado direito ϕ, como a resultante da alteração acima.

A solução que propomos é fazer-se a restrição da hipótese do teorema seguinte, que demonstramos ser condição suficiente para que este problema não ocorra. Note-se que a condição, embora não seja necessária e suficiente, é de fácil verificação.

Teorema 1: Seja G = (Ve, Vs, Vn, S, P) uma GTLC onde as produções de P são da forma:
$A ::= \alpha\gamma$, $\alpha \in (Ve \cup Vn)^*$, $\gamma \in (Vs \cup Ve)^*$
(ou seja, nenhum terminal de saída ocorre à esquerda de qualquer não-terminal).
Seja ω uma sentença de G1 (gramática de entrada).
Então a sequência obtida pela emissão dos símbolos de saída de cada produção no momento da redução correspondente em G1, durante o reconhecimento de ω, é igual à parte de saída da sentença de G cuja parte de entrada é ω.

Demonstração: Para atender à restrição as produções de G serão da forma $A ::= \alpha\gamma$,
onde $\alpha \in (Ve \cup Vn)^*$ e $\gamma \in (Vs \cup Ve)^*$.
Consideremos a derivação direita de uma sentença . Se esta é feita com uma única produção o resultado é provado trivialmente. Se mais de uma produção é empregada, a primeira, será da forma $S \Rightarrow \alpha C\gamma$, $\alpha \in (Ve \cup Vn)^*$, $\gamma \in (Vs \cup Ve)^*$, $C \in Vn$, isto é, C é o não terminal mais à esquerda na produção. A derivação direita de ω e a árvore correspondente serão da forma:

$S \Longrightarrow \alpha C \gamma \Longrightarrow \alpha\beta\delta\gamma \Longrightarrow \ldots \Longrightarrow \omega$, com $\beta \in (Ve \cup Vn)^*$ e $\delta \in (Vs \cup Ve)^*$.



Como a ordem das reduções em análise ascendente corresponde ao inverso da ordem das derivações à direita, podemos garantir que:

a) os terminais de saída de δ precederão os de γ;

b) os terminais de saída da sub-árvore β precederão os de δ;

c) os terminais de saída da sub-árvore α precederão os sub-árvore C;

d) e assim sucessivamente.

Isto corresponde à leitura das folhas terminais de saída da esquerda para a direita e logo à parte de saída da sentença .

c.q.d.

De acordo com esta proposta, a produção (1) de Gif seria modificada para

C.IF ::= IF.E.THEN C then

IF.E.THEN ::= if E then if

Este processo é especialmente atraente se a análise sintática é feita por matrizes de transição, uma vez que a gramática original é necessariamente modificada numa maneira muito semelhante à do exemplo acima. O processo de transformação de uma gramática de operadores em gramática de operadores aumentada (Gries, 6), pode ser facilmente adaptado a GTLCs de operadores (GTLCs em cujas produções não ocorrem não-terminais adjacentes) que satisfaçam à condição:

"uma cadeia de símbolos de saída deve seguir um terminal de entrada ou estar no final da produção".

A adaptação a gramáticas de tradução do algoritmo de transformação de gramática de operadores em gramática de operadores aumentada consiste em tratar o terminal de entrada e os terminais de saída que o seguem com um único símbolo terminal, isto é, o mesmo símbolo de entrada seguido de duas cadeias de símbolos de saída diferentes são vistos como símbolos diferentes. As cadeias de símbolos de saída que seguirem um não-terminal devem aparecer obrigatoriamente no final da produção, e nesta situação permanecem, não sendo considerados pelo algoritmo. A gramática de tradução resultante terá as cadeias de símbolos de saída no final das produções, satisfazendo protanto a hipótese do teorema anterior.

Cabe ainda observar que a gramática de tradução aumentada deixa de ser uma gramática auxiliar para a análise sintática, para ser a própria gramática sobre a qual a análise sintática é feita. Isto é, a saída do analisador sintático passa a ser uma sequência de produções da gramática aumentada, e não da gramática original. Caso isto não seja desejado, é preciso modificar-se a gramática original para que atenda às condições do teorema an-

terior antes de sua transformação para gramática aumentada, e a mencionada vantagem do método com relação à análise sintática por matrizes de transição não se verifica.

Teríamos as seguintes produções na gramática de tradução livre-de-contexto aumentada, provenientes da produção (1):

C.IF ::= IF.E.THEN C then

IF.E.THEN ::= IF E then if

IF ::= if

no qual if é uma cadeia de símbolos de saída considerada, juntamente com "then", como um único símbolo terminal, e "then" aparece no final da produção, não sendo sua posição modificada.

Notação: Os símbolos não-terminais estrelados, resultantes do processo de transformação da gramática de operadores em gramática aumentada, serão distiguidos por grifo dos símbolos não-terminais originais, como no exemplo acima.

II.1.3 - Gramática de Tradução Livre-de-Contexto para LPS

Os símbolos de saída na gramática de tradução livre-de-contexto para LPS são nomes de rotinas semânticas. A introdução destes símbolos na sintaxe original LPS foi feita observando-se em que pontos seriam necessárias ações semânticas, para gerar código ou armazenar em atributos semânticos informações a serem posteriormente utilizadas. Escrevemos em primeiro lugar uma GTLC de operadores, transformada a seguir em GTLC aumentada, na qual as produções atendem às condições do teorema 1, não havendo símbolos de saída antes de não-terminais.

Observação: A gramática seguinte foi modificada em relação à gramática LPS original, tendo sido substituído por "#" o símbolo "arroba".

Nas produções apresentadas adiante, estão presentes apenas as regras da gramática LPS total que foram modificadas pela introdução de símbolos de saída e algumas para outras que o conjunto fizesse sentido. Algumas poucas regras muito particulares da linguagem LPS e portanto de pouco interesse geral, foram suprimidas, embora envolvessem ações semânticas, para maior brevidade na exposição. Pelo mesmo motivo supusemos já executadas as ações que envolvam a tabela de símbolos, tais como pesquisa e inserção de identificadores, sendo estes considerados símbolos terminais de acordo com seu tipo (idvariável e idrotina por exemplo) nos comandos, ou o símbolo identificador nas declarações. As constantes numéricas também foram agrupadas num único símbolo terminal (constante), e os operadores das expressões simples representados pelos terminais opunário, oprel, opad, opmul e opdesloc, além de and, or e xor, estando o primeiro associado ao terminal de saída opunário e os demais a opbinário. Finalmente as características voltadas à comunicação entre módulos separadamente compilados, relativas às declarações "global" e "external" foram suprimidas, pelas mesmas razões expostas anteriomente.

Embora não esteja explícito na sintaxe, em vista do que acaba de ser dito acerca de identificadores e operaçpes aritméticas, as seguintes rotinas (símbolos da saída) possuem necessariamente um parâmetro:

-opbinário e opunário, cujo parâmetro especifica a operação aritmética e pode assumir os valores -, NOT, EXPS, EXPZ, HIGH, LOW para opunário e +, -, *, |, MOD, AND, OR, XOR, RTL, RTR, SHL, SAL, SAR, =, <>, <, > , <=, >=, '<', '>', '<=', '>=' para opbinário.

-idvariável, idrótulo, idchave, idrotina e idconstante, cujo parâmetro pode ser considerado como a cadeia de caracteres que define o identificador, mas cujo valor é na prática um ponteiro para o bloco de informações deste identificador na tabela de símbolos.

Gramática de Tradução Livre-de-Contexto LPS (de Operadores)

```
COMANDO::=BLOCO
         | COMANDO.CONDICIONAL
         | COMANDO.WHILE
         | COMANDO.REPEAT
         | COMANDO.FOR
         | COMANDO.CASE
         | DESVIO
         | RETORNO
         | CHAMADA.ROTINA
         | ATRIBUIÇÃO

COMANDO.CONDICIONAL::=if EXPRESSÃO then if while COMANDO then
| if EXPRESSÃO then if while COMANDO else then else COMANDO else

COMANDO.WHILE ::=
while $ rótulo EXPRESSÃO do if while COMANDO fim while

COMANDO.REPEAT::= repeat $ rótulo COMANDO goto
                | repeat $ rótulo COMANDO until EXPRESSÃO  until

COMANDO.FOR::=
| for VARIÁVEL := EXPRESSÃO atribuição for to 1 constante
          EXPRESSÃO do limite COMANDO fim for
| for VARIÁVEL := EXPRESSÃO atribuição for downto -1 constante
          EXPRESSÃO do limite COMANDO fim for
| for VARIÁVEL := EXPRESSÃO atribuição for step EXPRESSÃO
          until step EXPRESSÃO do limite COMANDO fim for

COMANDO.CASE ::=
case EXPRESSÃO of case begin SEQUÊNCIA.CASOS end fim case

SEQUÊNCIA.CASOS ::= COMANDO fim caso
                  | SEQUÊNCIA.CASOS ; COMANDO fim caso

DESVIO ::= goto idrótulo idrótulo goto
         |  goto idchave idchave ( EXPRESSÃO ) índice goto chave
```

```
RETORNO ::= return return

CHAMADA.ROTINA ::= idrotina idrotina chamada
| idrotina idrotina ( LISTA.PARÂMETROS.EFETIVOS ) chamada

LISTA.PARÂMETROS.EFETIVOS ::= EXPRESSÃO parâmetro
     | LISTA.PARÂMETROS.EFETIVOS , EXPRESSÃO parâmetro

ATRIBUIÇÃO ::= VARIAVEL ::= EXPRESSÃO atribuição comando

EXPRESSÃO ::= EXPRESSÃO.SIMPLES
            | EXPRESSÃO.CONDICIONAL
            | VARIÁVEL ::= EXPRESSÃO atribuição expressao

EXPRESSÃO.CONDICIONAL ::=
if EXPRESSÃO then if while EXPRESSÃO
             else then expr EXPRESSÃO else expr

EXPRESSÃO.SIMPLES ::= TERMO5
           | EXPRESSÃO.SIMPLES or TERMO5 opbinário
           | EXPRESSÃO.SIMPLES xor TERMO5 opbinário

TERMO5::= TERMO4
        | TERMO5 and TERMO4 opbinário

TERMO4::= TERMO3
        | TERMO4 oprel TERMO3 opbinário

TERMO3::= TERMO2
        | TERMO3 opad TERMO2 opbinário
        | opunario OPERANDO opunário

TERMO2::= TERMO1
        | TERMO2 opmul TERMO1 opbinário
TERMO1::= OPERANDO
        | TERMO1 opdesloc OPERANDO opbinário
```

```
OPERANDO::= constante constante
          | idconstante idconstante
          | VARIÁVEL
          | CHAMADA FUNÇÃO
          | # idvariável # variável
          | (EXPRESSÃO)

CHAMADA.FUNÇÃO::= idfunção idfunção chamada
| idfunção idfuncao ( LISTA.PARÂMETROS.EFETIVOS) chamada

VARIÁVEL::= idvariável idvariável
          | idvariável idvariável (EXPRESSÃO) índice
          | OPERANDO ↑ idvariável idvariável
          | OPERANDO ↑ idvariável idvariável (EXPRESSÃO)  índice

DECLARAÇÃO.ROTINA.INTERNA::=
CABEÇALHO.ROTINA ; recepção COMANDO return

CABEÇALHO.ROTINA::= procedure identificador decl rotina
| procedure identificador decl rotina
                    ( SEQUÊNCIA.ESPECIFICAÇÃO.PARÂMETROS )
| tipo tipo procedure identificador decl rotina
| tipo tipo procedure identificador decl rotina
                    SEQUÊNCIA.ESPECIFICAÇÃO.PARAMETROS )

SEQUÊNCIA.ESPECIFICAÇÃO.PARÂMETROS::= ESPECIFICAÇÃO.PARÂMETROS
|  SEQUÊNCIA.ESPECIFICAÇÃO.PARÂMETROS , ESPECIFICAÇÃO.PARÂMETROS

ESPECIFICAÇÃO.PARÂMETROS::= tipo tipo SEQUENCIA.PARÂMETROS

SEQUÊNCIA.PARÂMETROS::= identificador decl parm
          | SEQUÊNCIA.PARÂMETROS , identificador decl parm

BLOCO::= begin SEQUÊNCIA.COMANDOS end

SEQUÊNCIA.COMANDOS::= COMANDO
          | SEQUÊNCIA.COMANDOS ; COMANDO
```

Gramática de Tradução Livre-de-Contexto LPS (Aumentada)

```
COMANDO::= BLOCO
         | COMANDO.CONDICIONAL
         | COMANDO.WHILE
         | COMANDO.REPEAT
         | COMANDO.FOR
         | COMANDO.CASE
         | DESVIO
         | RETORNO
         | CHAMADA.ROTINA
         | ATRIBUIÇAÕ

COMANDO.CONDICIONAL::= IF.THEN COMANDO then
                     | IF.THEN.COM.ELSE COMANDO else

IF.THEN.COM.ELSE::= IF.THEN COMANDO else then else

IF.THEN::= IF EXPRESSÃO then if while

IF::= if

COMANDO.WHILE::= WHILE.DO COMANDO fim while

WHILE.DO::= WHILE EXPRESSÃO do if while

WHILE::= while $.rótulo

COMANDO.REPEAT::= REPEAT COMANDO goto
                | REPEAT.UNTIL EXPRESSÃO until

REPEAT.UNTIL::= REPEAT COMANDO until

REPEAT::= repeat $ rótulo

COMANDO.FOR::= FOR.STEP.UNTIL.DO COMANDO fim for
```

```
FOR.STEP.UNTIL.DO::= FOR.STEP.UNTIL EXPRESSÃO do limite
                   | FOR.STEP EXPRESSÃO do limite

FOR.STEP.UNTIL::= FOR.STEP EXPRESSÃO until step

FOR.STEP ::= FOR.ATRIB EXPRESSÃO step atribuição for
     | FOR.ATRIB EXPRESSÃO to atribuição for 1 constante
     | FOR.ATRIB EXPRESSÃO downto atribuição for -1 constante

FOR.ATRIB::= FOR VARIÁVEL

FOR::= for

COMANDO.CASE::= CASE.BEGIN.END

CASE.BEGIN.END::= CASE.BEGIN SEQUÊNCIA.CASOS end fim case

CASE.BEGIN::= CASE.OF begin

CASE.OF::= CASE EXPRESSÃO of case

CASE::= case

SEQUÊNCIA.CASOS::= COMANDO fim caso
          | SEQUÊNCIA.CASOS.PTVIRG COMANDO fim caso

SEQUÊNCIA.CASOS.PTVIRG::= SEQUÊNCIA.CASOS ;

DESVIO::= GOTO.RÓTULO goto rótulo
        | GOTO.CHAVE goto chave

GOTO.RÓTULO::= GOTO idrótulo idrótulo

GOTO.CHAVE::= GOTO.ID.CHAVE.ABRE EXPRESSÃO ) índice

GOTO.ID.CHAVE.ABRE::= GOTO.ID.CHAVE (

GOTO.ID.CHAVE::= GOTO idchave idchave
```

```
GOTO::= goto

RETORNO::= RETURN

RETURN::= return return

CHAMADA.ROTINA::= ID.ROTINA chamada rotina
        | ID.ROTINA.ABRE.FECHA chamada rotina

ID.ROTINA.ABRE.FECHA::=
     ID.ROTINA.ABRE LISTA.PARÂMETROS.EFETIVOS )

ID.ROTINA.ABRE::= ID.ROTINA (

ID.ROTINA::= idrotina idrotina

LISTA.PARÂMETROS.EFETIVOS::= EXPRESSÃO parâmetro
| LISTA.PARÂMETROS.EFETIVOS.VIRG EXPRESSÃO parâmetro

LISTA.PARÂMETROS.EFETIVOS.VIRG::= LISTA.PARÂMETROS.EFETIVOS ,

ATRIBUIÇÃO::= VARIÁVEL.ATRIB EXPRESSÃO atribuição comando

VARIÁVEL.ATRIB::= VARIÁVEL :=

EXPRESSÃO::= EXPRESSÃO.SIMPLES
           | EXPRESSÃO.CONDICIONAL
           | VARIÁVEL.ATRIB EXPRESSÃO atribuição expressão

EXPRESSÃO.CONDICIONAL::= IF.THEN.EX.ELSE EXPRESSÃO else expr

IF.THEN.EX.ELSE::= IF.THEN EXPRESSÃO else then expr

EXPRESSÃO.SIMPLES::= TERMO
           | EXS.OR TERMO5 opbinário
           | EXS.XOR TERMO5 opbinário
```

```
EXS.OR::= EXPRESSÃO.SIMPLES or

EXS.XOR::= EXPRESSÃO.SIMPLES xor

TERMO5::= TERMO4
        | TERMO5.AND TERMO4 opbinário

TERMO5.AND::= TERMO5 and

TERMO4::= TERMO3
        | TERMO4.OPREL TERMO3 opbinário

TERMO4.OPREL::= TERMO4 oprel

TERMO3::= TERMO2
        | TERMO3.OPAD TERMO2 opbinário
        | OPUNÁRIO OPERANDO opunário

TERMO3.OPAD::= TERMO3 opad

OPUNÁRIO::= opunário

TERMO2::= TERMO1
        | TERMO2.OPMUL TERMO1 opbinário

TERMO2.OPMUL::= TERMO2 opmul

TERMO1::= OPERANDO
        | TERMO1.OPDESLOC OPERANDO opbinário

TERMO1.OPDESLOC::= TERMO1 opdesloc

OPERANDO::= CONSTANTE
          | ID.CONSTANTE
          | VARIAVEL
          | CHAMADA.FUNÇÃO
          | SUST.ID.VARIAVEL
          | ABRE.EXPRESSÃO.FECHA
```

```
CONSTANTE::= constante constante

ID.CONSTANTE::= idconstante idconstante

SUST.ID.VARIÁVEL::= SUSTENIDO idvariável # variável

SUSTENIDO::= #

ABRE.EXPRESSÃO.FECHA::= ABRE EXPRESSÃO )

ABRE::= (

VARIÁVEL::= ID.VARIÁVEL
          | ID.VARIÁVEL.ABRE.FECHA
          | SETA.ID.VARIÁVEL
          | SETA.ID.VARIÁVEL.ABRE.FECHA

SETA.ID.VARIÁVEL.ABRE.FECHA::=
     SETA.ID.VARIÁVEL.ABRE EXPRESSÃO ) índice

SETA.ID.VARIÁVEL.ABRE::= SETA.ID.VARIÁVEL (

SETA.ID.VARIÁVEL::= OPERANDO.SETA idvariável idvariável

OPERANDO.SETA::= OPERANDO ↑

ID.VARIÁVEL.ABRE.FECHA::= ID.VARIÁVEL.ABRE EXPRESSÃO ) índice

ID.VARIÁVEL.ABRE::= ID.VARIAVEL (

ID.VARIÁVEL::= idvariável idvariável

CHAMADA.FUNÇÃO::= ID.FUNÇÃO chamada função
         | ID.FUNÇÃO.ABRE.FECHA chamada funcao

ID.FUNÇÃO.ABRE.FECHA::= ID.FUNÇÃO.ABRE LISTA.PARÂMETROS.EFETIVOS
```

```
ID.FUNÇÃO.ABRE::= ID.FUNCAO (

ID.FUNÇÃO::= idfunção idfunção

DECLARAÇÃO.ROTINA.INTERNA::= PROC.ID.PTVIRG COMANDO return
                           | TIPO.PROC.ID.PTVIRG COMANDO return

PROC.ID.PTVIRG::= PROC.ID ; recepção
                | PROC.ID.ABRE.FECHA ; recepção

PROC.ID.ABRE.FECHA::=
     PROC.ID.ABRE SEQUÊNCIA.ESPECIFICAÇÃO.PARÂMETROS )

PROC.ID.ABRE::= PROC.ID (

PROC.ID::= PROCEDURE identificador decl rotina

PROCEDURE::= procedure

TIPO.PROC.ID.PTVIRG::= TIPO.PROC.ID ; recepção
                     | TIPO.PROC.ID.ABRE.FECHA ; recepção

TIPO.PROC.ID.ABRE.FECHA::=
TIPO.PROC.ID.ABRE SEQUÊNCIA.ESPECIFICAÇÃO.PARÂMETROS)

TIPO.PROC.ID.ABRE::= TIPO.PROC.ID (

TIPO.PROC.ID::= TIPO.PROC identificador decl rotina

TIPO.PROC::= TIPO procedure

TIPO::= tipo tipo

SEQUÊNCIA.ESPECIFICAÇÃO.PARÂMETROS::= ESPECIFICAÇÃO.PARÂMETROS
|     SEQUÊNCIA.ESPECIFICAÇÃO.PARÂMETROS.PTVIRG     ESPECIFICA-
ÇÃO.PARÂMETROS

SEQUÊNCIA.ESPECIFICAÇÃO.PARÂMETROS.PTVIRG::=
```

SEQUÊNCIA.ESPECIFICAÇÃO.PARÂMETROS

ESPECIFICAÇÃO.PARÂMETROS::= TIPO SEQUÊNCIA.PARÂMETROS

SEQUÊNCIA.PARÂMETROS::= IDENTIFICADOR
| SEQ.PARMS.VIRG.IDENTIFICADOR

SEQ.PARMS.VIRG.IDENTIFICADOR::= SEQ.PARMS.VIRG identificador decl parm

SEQ.PARMS.VIRG::= SEQUENCIA.PARAMETROS ,

IDENTIFICADOR::= identificador decl parm

BLOCO::= BEGIN.END

BEGIN.END::= BEGIN SEQUÊNCIA.COMANDOS end

BEGIN::= begin

SEQUÊNCIA.COMANDOS::= COMANDO
| SEQUÊNCIA.COMANDOS.PTVIRG COMANDO

SEQUÊNCIA.COMANDOS.PTVIRG::= SEQUÊNCIA.COMANDOS ;

II.2 - Geração de Código Intermediário

Na seção anterior estudamos uma maneira de transformar o programa em uma sequência de chamadas a rotinas semânticas, o que constitui a primeira fase da geração de código. O código intermediário será gerado à medida que executamos ordenadamente estas rotinas. Este código, porém, pode depender do resultado de outras rotinas semânticas, que já terão sido executadas. Para armazenar tais resultados, definiremos o que sejam atributos semânticos, cujo cálculo será a segunda tarefa das rotinas semânticas.

Se desejamos gerar o código de máquina paralelamente à análise sintática, o cálculo dos atributos semânticos estará sujeito ao método usado. Veremos portanto como se comportam com analisadores sintáticos ascendentes o cálculo de atributos e a própria geração de código. Por fim, estudaremos a aplicação destas considerações aos compiladores LPS.

II.2.1 - Atributos Semânticos

A introdução de atributos semânticos numa gramática de tradução livre-de-contexto consiste em incluir nas suas produções símbolos de um novo conjunto Va (vocabulário de atributos), associados aos símbolos de saída. Esta associação destaca os atributos calculados pela rotina semântica representada pelo terminal. Um outro símbolo ainda é necessário para indicar a propagação dos atributos, podendo aparecer do lado direito das produções imediatamente após um símbolo de saída ou um não-terminal, fazendo com que seus atributos sejam transmitidos ao não-terminal à esquerda da produção. Isto equivale a dizer que o símbolo de propagação indica a síntese de atributos de alguns nós filhos para o nó pai na árvore de derivação na gramática de saída, também chamada árvore de rotinas semânticas.

Definição: Uma gramática de tradução livre-de-contexto com atributos (GTLCA) é uma 8-tupla (Ve, Vs, Vn, Va, S, !, P, A) onde

1 - Ve é um conjunto finito de símbolos de entrada, ou vocabulário de entrada.

2 - Vs é um conjunto finito de símbolos de saída, ou vocabulário de saída, tal que $Ve \cap Vs = \Phi$.

3 - Vn é um conjunto finito de símbolos não-terminais, tal que $Vn \cap Ve = \Phi$ e $Vn \cap Vs = \Phi$.

4 - Va é um conjunto finito de símbolos indicadores de atributos, ou vocabulário de atributos, tal que
$Ve \cap Va = \Phi$, $Vs \cap Va = \Phi$, $Vn \cap Va = \Phi$.

5 - S é o símbolo inicial de G, $S \in Vn$.

6 - ! é o símbolo indicador de propagação de atributos, $! \notin (Ve \cup Vs \cup Vn \cup Va)$.

7 - P é um conjunto finito de regras da forma $A ::= \alpha$, sendo $A \in Vn$ e $\alpha \in (Ve \cup Vs \cup Vn \cup Va \cup \{!\})^*$ tais que não ocorre ! após um elemento de Ve em α. Os elementos de P são denominados produções e a relação $A ::= \alpha$ lê-se A produz α.

8 - A é um conjunto finito de definições de atributos da forma $\underline{b} ::= B1, B2, \ldots, Bn$ onde $\underline{b} \in Vs$ e $Bi \in Va$, $1 <= i <= n$

Sendo G = (Ve, Vs, Vn, Va, S, !, P, A) uma GTLCA, as definições de deriva diretamente, deriva, forma sentencial, sentença, derivação, parte de entrada, parte de saída, tradução, gramática de entrada, gramática de saída e árvore de derivação são análogas às anteriormente apresentadas para GTLC, considerando-se o conjunto $(Vs \cup \{!\})$ nas definições para GTLCA correspondendo ao conjunto dos símbolos de saída nas definições dadas para GTLC. Assim sendo, parte de entrada de uma sentença de um GTLCA, por exemplo, é a cadeia obtida eliminando-se de todas as ocorrências de elementos do conjunto $(Vs \cup \{!\})$. Serão adotadas para as GTLCAs as mesmas normas já enunciadas em 2.1.1, para representação dos símbolos e das produções das GTLCs.

O cálculo de atributos propriamente dito é tarefa das rotinas associadas aos símbolos de saída e não faz parte da gramática. Sua definição se faz em forma de texto de programa ou em linguagem corrente, e normalmente deverá levar em conta a forma de armazenamento dos atributos pelo compilador. A especificação

do cálculo de atributos pode ser feita, por exemplo, associando-se os atributos aos símbolos envolvidos através de alguma convenção pré-estabelecida, a ser convertida depois em programa na linguagem de implementação do compilador. O armazenamento e propagação dos atributos, porém, pode ser feito pelo próprio tradutor, que passa a ter, na prática, essa nova função, além de fazer o reconhecimento sintático e chamar as rotinas semânticas.

II.2.2 - Cálculo de Atributos Semânticos com Análise Sintática Ascendente

Considerando que desejamos fazer o cálculo dos atributos semânticos durante o processo de reconhecimento sintático, devemos investigar as relações deste com aquele, e o faremos para analisadores sintáticos ascendentes.

Imaginemos que neste caso o armazenamento dos atributos se faça em um pilha para cada atributo, segundo a seguinte estratégia:

- cada vez que uma rotina semântica é chamada, seus atributos são empilhados após o retorno;
- cada vez que uma redução é feita, todos os atributos dos símbolos à direita são retirados da pilha e em seguida empilhados aqueles que possuíam indicação de propagação (!).

Desta forma, cada rotina semântica tem acesso a todos os atributos dos símbolos que a precedem na árvore de rotinas semânticas, ou seja, que estão à sua "esquerda" na árvore.

Supondo que esta é a estratégia adotada, e que se X ε Va, X1 é o topo da pilha de armazenamento de X, X2 é o subtopo e assim por diante, e que X0 é o valor do atributo X da rotina onde se encontrar sua determinação, examinemos o seguinte exemplo.

Exemplo 3: Cálculo do valor de uma expressão aritmética na forma infixa.

Gv = (Ve, Vs, Vn, Va, E, !, P, A)

Ve = {0, 1, ..., 9, +, -, *, /, (,)}

Vs = {0, 1, ..., 9, +, -, *, /, ini.número, fim.número}

Vn = {E, T, F, NÚMERO, DÍGITO, FIMNÚMERO}

Va = {V}

E é o símbolo inicial

! é o indicador de propagação

As regras de P são:

```
E::= T!
   | E + T + !
   | E - T - !

T::= F!
   | T * F * !
   | T / F / !

F::= NÚMERO !
   | ( E   )

NÚMERO::= DÍGITO ini.número fim.número !
        | DÍGITO ini.número FIMNÚMERO fim.número !

FIMNÚMERO::= DÍGITO dígito !
           | FIMNÚMERO DÍGITO dígito !

DÍGITO::= 0 0 !
        |1 1 !
        .
        .
        .
        |9 9 !
```

As regras de A são

0::=V
.
.
.
9::=V
+::=V
-::=V
*::=V
/::=V
ini.número::=V
fim.número::=V
dígito::=V

As rotinas semânticas são:

0	V0:=0;
1	V0:=1;
.	.
.	.
.	.
9	V0:=9;
ini.número	V0:=V1;
dígito	V0:=10 * V2 + V1
fim.número	V0:=V1
+	V0:=V2 + V1
-	V0:=V2 - V1
*	V0:=V2 * V1
/	V0:=V2 | V1

II.2.3 - Geração de Código com Análise Sintática Ascendente

Da mesma forma que o cálculo de atributos, a geração de código por uma rotina semântica pode dispor dos atributos que a precedem na árvore de rotinas semânticas.

Supondo a mesma estratégia de armazenamento e a mesma convenção de representação dos atributos pelas rotinas semânticas, vejamos um exemplo em que o objetivo é a geração de código para expressões aritméticas em uma máquina com pilha.

Exemplo 4: Geração de código para expressões aritméticas em máquina com pilha.

1 - Descrição da Máquina : Possui uma pilha cujo topo é R e seis intruções, todas com um endereço de memória como parâmetro.

CG	ENDEREÇO	empilha o valor em ENDEREÇO
A	ENDEREÇO	desempilha e armazena em ENDEREÇO
ADI	ENDEREÇO	R::= R + (ENDEREÇO)
SUB	ENDEREÇO	R::= R - (ENDEREÇO)
MUL	ENDEREÇO	R::= R * (ENDEREÇO)
DIV	ENDEREÇO	R::= R / (ENDEREÇO)

2- Supomos que ao ser chamada a rotina identificador já foi efetuada uma pesquisa na tabela de símbolos e a variável ENDER.ID aponta a entrada correspondente.

3 - A posição 0 (zero) de memória é reservada para auxiliar os cálculos.

4 - Após a execução do código o resultado será o único valor em-

pilhado.

5 - A gramática de tradução LCA é:

Gc = (Ve, Vs, Vn, Va, E, !, P, A)

Ve = a, b, ▬ z, +, -, *, |, (,)

Vs = identificador, $\underline{+}$, $\underline{-}$, $\underline{*}$, $\underline{/}$

Vn = E, T, F, IDENTIFICADOR, LETRA

Va = ENDER

E é o símbolo inicial

! é o símbolo de propagação

As regras de P são:

E::= T!
| E + T $\underline{+}$!
| E - T $\underline{-}$!

T::= F !
| T * F $\underline{*}$!
| T | F $\underline{|}$!

F::= IDENTIFICADOR !
(E !)

IDENTIFICADOR::= LETRA identificador !
| LETRA FIMID identificador !

FIMID::= LETRA
| FIMID LETRA

LETRA::= a |b|...|z

As regras de A são:

identificador::= ENDER
+::= ENDER
-::= ENDER
*::= ENDER
/::= ENDER

As rotinas semânticas são:

identificador	E:= ENDER.ID;
+	OPERA("ADI");
-	OPERA("SUB");
*	OPERA("MUL");
/	OPERA("DIV");

A rotina OPERA (string OP) é:

```
ENDER:= "%";                    & resultado na pilha
if ENDER1 = "%" then            & segundo operando na pilha
begin
    ENDER1:= 0;
    GERA ("A", ENDER1);
end;
if ENDER2    "%" then           & primeiro operando na memória
   GERA ("CG", ENDER2);
GERA (OP, ENDER1);
```

A rotina GERA (string INSTR; word ENDER) emite a instrução INSTR com parâmetro ENDER.

Para a sentença de entrada a - b * c apresentamos a situação do gerador de código a cada redução

Redução pela Produção	Rotina Chamada	Código Emitido	Pilha do Atributo ENDER
13			
9	identificador		#a
7			#a
4			#a
1			#a
13			#a
9	identificador		#a, #b
7			#a, #b
4			#a, #b
13			#a, #b
9	identificador		#a, #b, #c
7			#a, #b, #c
5	*	CG b MUL c	#a, %
3	-	A 0 CG a SUB 0	

II.2.4 - Atributos Semânticos para o Compilador LPS

Sendo o analisador sintático do compilador LPS ascendente e sendo análise sintática e geração de código simultâneas, aplicam-se as observações anteriores quanto ao armazenamento dos atributos e seu acesso pelas rotinas semânticas. Também por este último motivo, os endereços de memória integrantes dos atributos referem-se ao próprio programa objeto. São cinco os atributos

semânticos para o compilador LPS: DADO, ROT, DROT, PROG e TIPO.

Atributo DADO: Descreve a localização dinâmica e o tipo de um dado manipulado pelo compilador.

Atributo ROT: Usado na compilação de chamadas de procedimento. Contém características tais como endereço de execução, número e tipos dos parâmetros, tipo da resposta (para função) e número de parâmetros já compilados.

Atributo DROT: Usado na compilação de declarações de procedimento. Contém características tais como endereço e tipo da resposta (para função), número e tipos dos parâmetros.

Atributo PROG: Refere-se a um endereço, definido ou não, no programa objeto

Atributo TIPO: Guarda o tipo de uma declaração (byte ou word).

Os quatro primeiros atributos precisam ser guardados em pilhas, mas o atributo TIPO só precisa de uma variável para armazenar seu valor. A descrição destes atributos, em termos dos campos de que se compõem, encontra-se no apêndice. A descrição da forma com são tratados pelas rotinas semânticas será vista no capítulo seguinte. Por ora apresentaremos apenas uma GTLCA para o compilador LPS, sujeita às mesmas limitações da GTLC dada em 2.1.3. As produções modificadas foram aquelas onde atributos são transmitidos ao não-terminal à esquerda, nas quais passa a figurar o sinal de propagação de atributos.

Gramática de Tradução Livre-de-Contexto com Atributos LPS

1 - Produções

```
COMANDO::= BLOCO
         | COMANDO.CONDICIONAL
         | COMANDO.WHILE
         | COMANDO.REPEAT
         | COMANDO.FOR
         | COMANDO.CASE
         | DESVIO
         | RETORNO
         | CHAMADA.ROTINA
         | ATRIBUIÇAO

COMANDO.CONDICIONAL::= IF.THEN COMANDO then
                     | IF.THEN.COM.ELSE COMANDO else

IF.THEN.COM.ELSE::= IF.THEN COMANDO else then else !

IF.THEN::= IF EXPRESSÃO then if while !

IF::= if

COMANDO.WHILE::= WHILE.DO COMANDO fim while

WHILE.DO::= WHILE ! EXPRESSÃO do if while !

WHILE::= while $ rótulo !

COMANDO.REPEAT::= REPEAT COMANDO goto
                | REPEAT.UNTIL EXPRESSÃO until

REPEAT.UNTIL::= REPEAT ! COMANDO until

REPEAT::= repeat $ rótulo !
```

```
COMANDO.FOR::= FOR.STEP.UNTIL.DO COMANDO fim for

FOR.STEP.UNTIL.DO::= FOR.STEP.UNTIL ! EXPRESSÃO do limite !
                   | FOR.STEP ! EXPRESSÃO do limite !

FOR.STEP.UNTIL::= FOR.STEP ! EXPRESSÃO until step !

FOR.STEP ::= FOR.ATRIB EXPRESSÃO step atribuição for !
| FOR.ATRIB EXPRESSÃO to atribuição for ! 1 constante !
| FOR.ATRIB EXPRESSÃO downto atribuição for ! -1 constante !

FOR.ATRIB::= FOR VARIÁVEL

FOR::= for

COMANDO.CASE::= CASE.BEGIN.END

CASE.BEGIN.END::= CASE.BEGIN SEQUÊNCIA.CASOS end fim case

CASE.BEGIN::= CASE.OF ! begin

CASE.OF::= CASE EXPRESSÃO of case !

CASE::= case

SEQUÊNCIA.CASOS::= COMANDO fim caso !
           | SEQUÊNCIA.CASOS.PTVIRG ! COMANDO fim caso !

SEQUÊNCIA.CASOS.PTVIRG::= SEQUÊNCIA.CASOS ! ;

DESVIO::= GOTO.RÓTULO goto rótulo
        | GOTO.CHAVE goto chave

GOTO.RÓTULO::= GOTO idrótulo idrótulo !

GOTO.CHAVE::= GOTO.ID.CHAVE.ABRE EXPRESSÃO ) índice !

GOTO.ID.CHAVE.ABRE::= GOTO.ID.CHAVE ! (
```

```
GOTO.ID.CHAVE::= GOTO idchave idchave !

GOTO::= goto

RETORNO::= RETURN

RETURN::= return return

CHAMADA.ROTINA::= ID.ROTINA chamada rotina
        | ID.ROTINA.ABRE.FECHA chamada rotina

ID.ROTINA.ABRE.FECHA::=
     ID.ROTINA.ABRE ! LISTA.PARÂMETROS.EFETIVOS )

ID.ROTINA.ABRE::= ID.ROTINA ! (

ID.ROTINA::= idrotina idrotina !

LISTA.PARÂMETROS.EFETIVOS::= EXPRESSÃO parâmetro
     | LISTA.PARÂMETROS.EFETIVOS.VIRG EXPRESSÃO parâmetro

LISTA.PARÂMETROS.EFETIVOS.VIRG::= LISTA.PARÂMETROS.EFETIVOS ,

ATRIBUIÇÃO::= VARIÁVEL.ATRIB EXPRESSÃO atribuição comando

VARIÁVEL.ATRIB::= VARIÁVEL ! :=

EXPRESSÃO::= EXPRESSÃO.SIMPLES !
           | EXPRESSÃO.CONDICIONAL !
           | VARIÁVEL.ATRIB EXPRESSÃO atribuição expressão !

EXPRESSÃO.CONDICIONAL::= IF.THEN.EX.ELSE EXPRESSÃO else expr !

IF.THEN.EX.ELSE::= IF.THEN EXPRESSÃO else then expr !

EXPRESSÃO.SIMPLES::= TERMO !
           | EXS.OR TERMO5 opbinário !
```

```
            | EXS.XOR TERMO5 opbinário !

EXS.OR::= EXPRESSÃO.SIMPLES ! or

EXS.XOR::= EXPRESSÃO.SIMPLES ! xor

TERMO5::= TERMO4 !
| TERMO5.AND TERMO4 opbinário !

TERMO5.AND::= TERMO5 ! and

TERMO4::= TERMO3 !
        | TERMO4.OPREL TERMO3 opbinário !

TERMO4.OPREL::= TERMO4 ! oprel

TERMO3::= TERMO2 !
        | TERMO3.OPAD TERMO2 opbinário !
        | OPUNÁRIO OPERANDO opunário !

TERMO3.OPAD::= TERMO3 ! opad

OPUNÁRIO::= opunário

TERMO2::= TERMO1 !
        | TERMO2.OPMUL TERMO1 opbinário !

TERMO2.OPMUL::= TERMO2 ! opmul

TERMO1::= OPERANDO
        | TERMO1.OPDESLOC OPERANDO opbinário !

TERMO1.OPDESLOC::= TERMO1 ! opdesloc

OPERANDO::= CONSTANTE !
          | ID.CONSTANTE !
          | VARIAVEL !
          | CHAMADA.FUNÇÃO !
```

```
          | SUST.ID.VARIAVEL !
          | ABRE.EXPRESSÃO.FECHA !

CONSTANTE::= constante constante !

ID.CONSTANTE::= idconstante idconstante !

SUST.ID.VARIÁVEL::= SUSTENIDO idvariável # variável !

SUSTENIDO::= #

ABRE.EXPRESSÃO.FECHA::= ABRE EXPRESSÃO !)

ABRE::= (

VARIÁVEL::= ID.VARIÁVEL !
          | ID.VARIÁVEL.ABRE.FECHA !
          | SETA.ID.VARIÁVEL !
          | SETA.ID.VARIÁVEL.ABRE.FECHA !

SETA.ID.VARIÁVEL.ABRE.FECHA::=
     SETA.ID.VARIÁVEL.ABRE EXPRESSÃO ) índice !

SETA.ID.VARIÁVEL.ABRE::= SETA.ID.VARIÁVEL ! (

SETA.ID.VARIÁVEL::= OPERANDO.SETA idvariável idvariável !

OPERANDO.SETA::= OPERANDO ! ↑

ID.VARIÁVEL.ABRE.FECHA::= ID.VARIÁVEL.ABRE EXPRESSÃO índice !

ID.VARIÁVEL.ABRE::= ID.VARIAVEL ! (

ID.VARIÁVEL::= idvariável idvariável !

CHAMADA.FUNÇÃO::= ID.FUNÇÃO chamada função !
         | ID.FUNCAO.ABRE.FECHA chamada funcao !
```

```
ID.FUNÇÃO.ABRE.FECHA::=
     ID.FUNÇÃO.ABRE ! LISTA.PARÂMETROS.EFETIVOS

ID.FUNÇÃO.ABRE::= ID.FUNCAO ! (

ID.FUNÇÃO::= idfunção idfunção !

DECLARAÇÃO.ROTINA.INTERNA::= PROC.ID.PTVIRG COMANDO return
                           | TIPO.PROC.ID.PTVIRG COMANDO return

PROC.ID.PTVIRG::= PROC.ID ! ; recepção
                | PROC.ID.ABRE.FECHA ! ; recepção

PROC.ID.ABRE.FECHA::=
PROC.ID.ABRE ! SEQUÊNCIA.ESPECIFICAÇÃO.PARÂMETROS )

PROC.ID.ABRE::= PROC.ID ! (

PROC.ID::= PROCEDURE identificador decl rotina !

PROCEDURE::= procedure

TIPO.PROC.ID.PTVIRG::= TIPO.PROC.ID ! ; recepção
                     | TIPO.PROC.ID.ABRE.FECHA ! ; recepção

TIPO.PROC.ID.ABRE.FECHA::=
TIPO.PROC.ID.ABRE ! SEQUÊNCIA.ESPECIFICAÇÃO.PARÂMETROS)

TIPO.PROC.ID.ABRE::= TIPO.PROC.ID ! (

TIPO.PROC.ID::= TIPO.PROC identificador decl rotina !

TIPO.PROC::= TIPO ! procedure

TIPO::= tipo tipo !
```

```
SEQUÊNCIA.ESPECIFICAÇÃO.PARÂMETROS::= ESPECIFICAÇÃO.PARÂMETROS
|      SEQUÊNCIA.ESPECIFICAÇÃO.PARÂMETROS.PTVIRG      ESPECIFICA-
ÇÃO.PARÂMETROS

SEQUÊNCIA.ESPECIFICAÇÃO.PARÂMETROS.PTVIRG::=
SEQUÊNCIA.ESPECIFICAÇÃO.PARÂMETROS

ESPECIFICAÇÃO.PARÂMETROS::= TIPO SEQUÊNCIA.PARÂMETROS

SEQUÊNCIA.PARÂMETROS::= IDENTIFICADOR
                      | SEQ.PARMS.VIRG.IDENTIFICADOR

SEQ.PARMS.VIRG.IDENTIFICADOR::=   SEQ.PARMS.VIRG   identificador
decl parm

SEQ.PARMS.VIRG::= SEQUENCIA.PARAMETROS ,

IDENTIFICADOR::= identificador decl parm

BLOCO::= BEGIN.END

BEGIN.END::= BEGIN SEQUÊNCIA.COMANDOS end

BEGIN::= begin

SEQUÊNCIA.COMANDOS::= COMANDO
                    | SEQUÊNCIA.COMANDOS.PTVIRG COMANDO

SEQUÊNCIA.COMANDOS.PTVIRG::= SEQUÊNCIA.COMANDOS ;
```

2 - Regras de Definição de Atributos

then else::= PROG
if while ::= PROG
$ rótulo::= PROG
limite::= DADO,PROG,PROG
step::= DADO
atribuição for::= DADO
1 constante::= DADO
-1 constante::= DADO
fim caso::= PROG
case::= PROG, PROG
idrótulo::=PROG
idchave::= DADO
idrotina::= ROT
índice::= DADO
atribuição expressão::= DADO
then expr::= DADO, PROG
else expr::= DADO
opbinário::= DADO
opunário::= DADO
constante::= DADO
idconstante::= DADO
variável::= DADO
idvariável::= DADO
chamada função::= DADO
idfunção::= ROT
declrotina::= DROT
tipo::= TIPO

II.3 - Código Intermediário para LPS

As instruções intermediárias LPS têm o formato geral de uma tripla (operação, parâmetro 1, parâmetro 2), cujos parâmetros são da forma dos atributos semânticos. Uma vez que a compilação é feita em um só passo, os endereços de memória integrantes das instruções referem-se ao próprio código objeto. Isto é, os endereços de variáveis, de execução de procedimento, de desvio, etc, referidos pelas instruções intermediárias atráves de seus parâmetros, são endereços "definitivos", na memória do processador objeto - o símbolo $ refere-se ao próximo endereço livre na área de programa, e o símbolo £ ao próximo endereço livre na área de dados. Desta forma, a arquitetura auxiliar para o código intermediário consta apenas de duas pilhas e um registrador de estado. Segue-se uma descrição sucinta desta arquitetura e do conjunto de instruções intermediárias.

a)**Pilha de Operandos (PO)**: É usada no cálculo de expressões, passagem de parâmetros para procedimentos e resposta das funções. O resultado das instruções intermediárias que realizam operações aritméticas e uma série de outras é fornecido no topo desta pilha. Além disso, uma vez que o atributo DADO pode referir-se a um seu elemento, topo e subtopo da PO podem ser parâmetros das instruções intermediárias. Neste caso, serão retirados da pilha durante a execução da instrução, antes do empilhamento do resultado.

b)**Pilha de Ligação (PL)**: É usada para armazenar os endereços de retorno dos procedimentos . Algumas intruções operam implicitamente sobre a PL, acrescentando, consultando ou retirando o elemento do topo.

c)**Registrador de Estado (RE)**: É consultado pelas intruções de desvio condicional, podendo assumir os valores "falso e "verdadeiro". Pode ser modificado pela instrução VL.

d)**Conjunto de Instruções**: Descrevemos a seguir o conjunto de

instruções intermediárias, representadas por triplas (operação, parâmetro1, parâmetro2). No lugar dos parâmetros indicamos o nome do atributo (ou atributos) cuja forma possuem. Se em vez de um nome de atributo escrevemos "-", é porque este parâmetro não é levado em conta, ou seja, a instrução tem um ou nenhum parâmetro.

d.1)**Instruções com um parâmetro do tipo DADO:** Excetuando-se "VL", "DCG" e "DVD" estas instruções produzem um resultado que é acrescentado à pilha de operandos. Neste caso, apenas o resultado está descrito. Se o parâmetro refere-se à PO, será retirado antes do empilhamento do resultado.

(CGE, DADO, -) : endereço do parâmetro

(CG, DADO, -): valor do parâmetro

(EXPS, DADO, -): valor do parâmetro expandido para word com propagação de sinal

(EXPZ, DADO, -): valor do parâmetro expandido para word com propagação de zeros

(LOW, DADO, -): valor do byte menos significativo do parâmetro

(HIGH, DADO, -): valor do byte mais significativo do parâmetro

(NEG, DADO, -): negação do parâmetro (complemento a 2)

(NOT, DADO, -): complemento a 1 do parâmetro

(VL, DADO, -): o registrador de estado recebe o valor lógico "verdadeiro" se o valor do parâmetro é ímpar e "falso" caso contrário

(DVD, DADO, -): o fluxo de execução é desviado para o endereço de memória dado pelo valor do parâmetro

(DCG, DADO, -): o topo da pilha de operandos é retirado, e armazenado no endereço do parâmetro

d.2) **Instruções com dois parâmetros do tipo DADO:** Primeiramente as instruções relativas a operações aritméticas, lógicas e de deslocamento. Estas instruções produzem um resultado que é acrescentado à pilha de operandos, após a retirada dos elementos associados aos parâmetros que porventura houvesse. Indicamos apenas o valor do resultado, sendo seu tipo igual ao tipo da instrução.

(ADI, DADO1, DADO2): DADO1 + DADO2

(SUB, DADO1, DADO2): DADO1 - DADO2

(MUL, DADO1, DADO2): DADO1 * DADO2

(DIV, DADO1, DADO2): DADO1 / DADO2

(MOD, DADO1, DADO2): DADO1 mod DADO2

(E, DADO1, DADO2): DADO1 and DADO2

(OU, DADO1, DADO2): DADO1 or DADO2

(OX, DADO1, DADO2): DADO1 xor DADO2

(GE, DADO1, DADO2): DADO1 rtl DADO2

(GD, DADO1, DADO2): DADO1 rtr DADO2

(DLE, DADO1, DADO2): DADO1 shl DADO2

(DLD, DADO1, DADO2): DADO1 shr DADO2

(DAE, DADO1, DADO2): DADO1 sal DADO2

(DAD, DAD01, DAD02): DAD01 sar DAD02

O próximo grupo de instruções produz resultado zero se a condição indicada for falsa, e um se for verdadeira. Este resultado é acrescentado a pilha de operandos, após a retirada dos elementos associados aos parâmetros.

(IGU , DAD01, DAD02): DAD01 = DAD02

(DIF , DAD01, DAD02): DAD01 <> DAD02

(MEN , DAD01, DAD02): DAD01 < DAD02

(MAI , DAD01, DAD02): DAD01 > DAD02

(MEG , DAD01, DAD02): DAD01 <= DAD02

(MAG , DAD01, DAD02): DAD01 >= DAD02

(MENL, DAD01, DAD02): DAD01 '<' DAD02

(MAIL, DAD01, DAD02): DAD01 '>' DAD02

(MEGL, DAD01, DAD02): DAD01 '<=' DAD02

(MAGL, DAD01, DAD02): DAD01 '>=' DAD02

As próximas instruções armazenam no endereço do primeiro parâmetro o valor do segundo, retirando da pilha de operandos os elementos associados aos parâmetros. Apenas a primeira produz um resultado, acrescentado à mesma pilha, que é o próprio valor do segundo operando convertido ao tipo do primeiro.

(ARM , DAD01, DAD02)

(ARMC, DAD01, DAD02)

d.3)**Instruções em que um parâmetro é do tipo PROG**

(DV, PROG, -): o fluxo de execução é desviado para o endereço dado por PROG

(DVF, PROG, -): idem, se o registrador de estado tem o valor "falso"

(DVI, PROG, DADO) : o fluxo de execução é desviado para o endereço contido na posição de memória dada por PROG indexado de DADO

(DW, PROG,): corresponde a uma palavra com o valor do parâmetro.

d.4)**Instrução com um parâmetro do tipo ROT**: São as instruções de chamada de procedimento. O endereço de retorno é acrescentado à pilha de ligação, e o fluxo de execução desviado para o endereço de execução dado pelo atributo ROT. Além disso, na instrução seguinte a "CHF" a pilha de operandos terá mais um elemento, correspondente ao valor de retorno da função.

(CHS, ROT, -): chamada de rotina

(CHF, ROT, -): chamada de função

d.5)**Outras instruções:**

(RET, -, -): retira o elemento do topo da pilha de ligação e desvia a execução para o endereço por ele indicado

(VAG, número, -): cria um espaço na área de dados com tamanho em bytes igual a "número".

(DEF, número, -): associa "número" ao endereço corrente de programa

III - Rotinas Semânticas do Compilador LPS

O objetivo deste capítulo é descrever a maneira pela qual as rotinas semânticas, resultantes do processo de tradução desenvolvido anteriormente, são capazes de gerar código para as principais construções que, de uma maneira geral, constituem as linguagens de programação correntes. Ao invés de estudar cada problema isoladamente, com o auxílio de um gramática-exemplo, estudaremos todos no contexto da gramática de tradução livre-de-contexto com atributos do compilador LPS.

Esta atitude nos permitirá uma melhor visão global da tarefa a ser desempenhada pelas rotinas semânticas, mantendo-se suficientemente geral, já que a linguagem LPS inclui as construções normalmente encontradas em linguagens de programação. Além disso, teremos a certeza de que os ítens discutidos comporão um todo coerente, visto terem dado origem a um compilador já largamente empregado.

Dividimos em três grupos as rotinas semânticas a serem estudadas para que as interrelações se evidenciassem. O primeiro é o grupo das rotinas chamadas durante a compilação das expressões e comandos de atribuição, que se relacionam a operandos ou operações entre eles.

No segundo estão as rotinas ligadas à compilação dos demais comandos, apresentadas em conjunto as responsáveis por cada um destes.

As rotinas usadas para resolver a declaração de procedimentos e funções, que apresentam algumas características especiais, formam o terceiro grupo.

Nos exemplos apresentados a seguir, utilizamos algumas vezes uma representação sintética dos atributos DADO, ROT e PROG através de seu campo LOC. Uma vez que os parâmetros das instruções e os elementos das pilhas de atributos tem formato bem definido, descrito no apêndice, o campo LOC identifica o atributo quando os valores dos demais campos já estiverem claros, tendo em vista sua representação em outro ponto do exemplo ou a analogia com exemplo anteriores. O símbolo $ indica o próximo endereço livre na área de programa, e o símbolo £ o mesmo na área de

dados.

III.1 - Expressões e Comando de Atribuição

A atribuição do resultado de uma expressão a uma variável é vista pelo gerador de código do compilador LPS como uma operação binária que armazena no endereço do primeiro operando o valor do segundo operando. Por esta razão expressões-simples, expressões-atribuição e comandos-atribuição são tratados de maneira muito semelhante. Conforme indica a sintaxe de tradução livre-de-contexto com atributos do compilador LPS, são chamadas rotinas semânticas para cada operando e em seguida a rotina que resolve a operação (notação polonesa pós-fixa). Já para expressões condicionais são necessárias ainda outras rotinas semânticas.

A chamada à rotina associada a um operador se faz de acordo com sua posição numa escala de prioridade, sendo o operador passado como parâmetro para uma das rotinas opbinário ou opunário. Não havendo parênteses, as chamadas vinculadas às operações mais prioritárias são feitas antes, prevalecendo a operação mais à esquerda em caso de igual prioridade. A ordem decrescente de prioridade das operações admitidas em LPS e o símbolo de entrada que representa cada conjunto de operadores de mesma prioridade são apresentados na tabela 1.

Do ponto de vista das rotinas opunário e opbinário esta ordem de precedência é irrelevante. O importante é notar que, uma vez que todos os símbolos de saída GTLCA que encerram a análise de um operando possuem o atributo DADO, que descreverá o acesso a este operando, e tendo sido as expressões traduzidas para a forma pós-fixa, a rotina opbinário encontrará no sub-topo e topo da pilha de DADO os atributos relativos ao primeiro e segundo operandos, respectivamente. A rotina opunário, por sua vez, encontrará no topo da mesma pilha o atributo DADO de seu operando.

OPERADORES	PRIORIDADE	SÍMBOLO DE ENTRADA
unários: +, -, NOT, EXPS, EXPZ, HIGH, LOW	7	opunário
deslocamento: RTL, RTR, SHL, SHR, SAL, SAR	6	opdeslocamento
multiplicativos: *, /, MOD	5	opmultiplicativo
aditivos: +, -	4	opaditivo
relação: comparações aritméticas e lógicas	3	oprelação
AND	2	and
OR, XOR	1	or, xor
:=	0	:=

Tabela 1 - Prioridade de Operadores

III.1.1 - Operandos

Como podemos observar na GTLCA, as rotinas associadas aos símbolos de saída constante, idconstante, # variável, idvariável e chamada função (vide também tabela 1) são as responsáveis pelo cálculo do atributo DADO associado ao operando representado por um identificador ou constante na expressão. Representam portanto a interface entre os dados da linguagem e sua representação pelo compilador, no atributo DADO, devendo eliminar os casos que não se deseje representar, transformando-os em outros, inclusive a-

tráves da emissão de algum código.

Podemos, por exemplo, reduzir a um mínimo a variedade de formas assumíveis pelo atributo DADO, gerando a cada chamada de uma das rotinas acima o código necessário para carregar na pilha de operandos o valor de cada operando que não esteja à esquerda de um sinal de atribuição, ou seu endereço caso contrário. Desta forma o atributo DADO precisará indicar apenas o tipo do dado e se está na pilha seu valor ou endereço. Não obstante sua simplicidade esta opção tem a desvantagem de esgotar rapidamente os registradores que porventura implementem a pilha de operandos, provocando gasto de código para movimentar os dados para outras localizações.

O outro extremo consiste em representar os dados em todas as formas em que se apresentem nas expressões, o que não importará em gasto adicional de código qualquer que seja a implementação da pilha de operandos. A representação interna dos dados para o gerador de código torna-se bem mais complexa, mas qualquer outra solução pode revelar-se ineficiente dependendo do processador-objeto, e esta é portanto a opção mais aconselhável se desejamos manter a independência quanto ao processador até o código intermediário, e será portanto a que escolheremos.

Se, por outro lado, considerarmos um determinado processador-objeto, podemos optar por alguma solução entre as duas precedentes, fazendo nas rotinas de tratamento dos operandos (constante, idconstante, etc.) as transformações que, sendo de qualquer forma necessárias devido às limitações do jogo de instruções da máquina, não importem em uso adicional de registradores, ou mesmo aquelas em que seja necessário alocar um novo registrador, desde que não haja movimentações de dados desnecessários em expressões usuais.

Nos exemplos seguintes procuramos exibir o comportamento das rotinas de tratamento dos diversos tipos de operandos das expressões da linguagem LPS.

Exemplo 1:

```
constant C=4; word W,X;
byte R ref W; word RD pos R+C;
byte RA ref *;
```

entrada	última roti-chamada	descrição do atributo DADO calculado
X	idvariável	LOC=#X TIPO=word INDIR=ender REL=rd DESLOC=0 INDICE=não
R	idvariável	LOC=#W TIPO=byte INDIR=ref REL=rd DESLOC=0 INDICE=não
RD	idvariável	LOC=#W TIPO=word INDIR=ref REL=rd DESLOC=4 INDICE=não
W↑RA	idvariável	LOC=#W TIPO=byte INDIR=ref REL=rd DESLOC=0 INDICE=não
(W+X)↑RA	idvariável	LOC=% TIPO=byte INDIR=ender REL=-- DESLOC=-- INDICE=não
C	idconstante	LOC=4 TIPO=byte INDIR=valor REL=abs DESLOC=0 INDICE=não
100	idconstante	LOC=100 TIPO=word INDIR=valor REL=abs DESLOC=0 INDICE=não
#X	# variável	LOC=#X TIPO=word INDIR=valor REL=rd DESLOC=0 INDICE=não

Observações:

1) Uma variável "ref *" apontada por uma variável "word" não re-

ferida é considerada como variável referida àquela que a aponta. Por isso o atributo DADO para as entradas R e W RA é idêntico.
2) Quando uma variável "ref *" estiver apontada por qualquer outro OPERANDO, seu endereço será acrescentado à pilha de operandos (LOC=%, INDIR=ender).
3) Os símbolos de saída constante e # variável indicam também chamadas à rotina id constante. O atributo resultante terá no campo LOC o próprio valor do operando, e não seu endereço como nos demais casos (INDIR=valor).
4) Se o endereço ou o valor de um dado está na pilha de operandos (LOC=%), os campos REL e DESLOC do atributo DADO associado são irrelevantes.

Exemplo 2:

```
word X,Y;
byte(10) VB;
word(10) VW;
```

2.a) Entrada: VW(VB(X))

rotina chamada	pilha do atributo DADO ---->
idvariável	LOC =#VW TIPO=word INDIR=ender REL=rd DESLOC=0 INDICE=não
idvariável	LOC=#VB TIPO=byte INDIR=ender REL=rd DESLOC=0 INDICE=não
idvariável	LOC=#X TIPO=word INDIR=ender REL=rd DESLOC=0 INDICE=não
índice	LOC=#VB TIPO=byte INDIR=ender REL=rd DESLOC=0 INDICE=não LOC=#X TIPO=word INDIR=ender REL=rd DESLOC=0 INDICE=sim
índice	LOC=#VW TIPO=word INDIR=ender REL=rd DESLOC=0 INDICE=não LOC=#VB TIPO=word INDIR=ender REL=rd DESLOC=0 INDICE=sim LOC=#X TIPO=word INDIR=ender REL=rd DESLOC=0 INDICE=sim

2.b) Entrada: VW (X+Y)

rotina chamada	código emitido	pilha do atributo DADO --->
idvariável		LOC=#VW TIPO=word INDIR=ender REL=rd DESLOC=0 INDICE=não
idvariável idvariável opbinário	(ADI,X,Y)	LOC=% TIPO=word INDIR=valor REL=-- DESLOC=-- INDICE=não
índice		LOC=VW TIPO=word INDIR=ender REL=rd DESLOC=0 INDICE=não LOC=% TIPO=word INDIR=valor REL=-- DESLOC=-- INDICE=sim

Observações:

1) Nenhum código é emitido pela rotina índice. O atributo DADO calculado pela rotina índice pode ser representado por VW(VB(X)) no exemplo 2.a) e por VW(%) no exemplo 2.b).

Exemplo 3:

```
byte procedure F; begin end;
```

entrada	última rotina chamada	código emitido	descrição do atributo DADO resultante
F	chamada função	(CHF,f,-)	LOC=% TIPO=byte INDIR=valor REL=-- DESLOC=-- INDICE=não

Observações:

1) Uma função é sempre imediatamente executada, portanto a rotina chamada função sempre produz um atributo DADO com LOC=%.

III.1.2 - Operações Unárias e Binárias

São opunário e opbinário as rotinas semânticas responsáveis pela geração do código intermediário que realiza as operações integrantes de uma expressão simples. Seu trabalho está bastante simplificado uma vez que a tarefa de preparação dos operandos para a operação segundo as instruções disponíveis no processador foi atribuída à próxima fase da compilação, e os operandos das instruções intermediárias coincidem com os próprios atributos do tipo DADO. De acordo com a GTLCA, ao resultado também corresponderá um atributo DADO o qual, em virtude da instrução intermediária gerada; terá os campos LOC, INDIR, e INDICE iguais a "%", "valor" e "não", respectivamente. Sendo neste caso os campos REL e DESLOC irrelevantes, resta apenas a determinação do TIPO, segundo as seguintes regras, dependendo da operação recebida como parâmetro:

a) operações binárias aritméticas e lógicas (+, -, *, /, MOD,

AND, OR, XOR). O TIPO do resultado é "byte" se e somente se os dois operandos têm TIPO igual a "byte", e "word" caso contrário.

b) operações de deslocamento (RTL, RTR, SHL, SHR, SAL, SAR): o TIPO do resultado é sempre o TIPO do primeiro operando.

c) operações de relação (=, <>, <, >, <=, >=, '<=', '>=', '<', '>'): o TIPO do resultado é sempre "byte".

d) operações unárias aritméticas e lógicas (+, -, NOT): o TIPO do resultado é sempre igual ao TIPO do operando.

e) operações de conversão (EXPS, EXPZ, HIGH, LOW): o TIPO do resultado é "word" para EXPS, EXPZ e "byte" para as demais.

Além de determinar o campo TIPO do atributo resultado, as duas rotinas devem emitir a instrução indetemediária relativa à operação corrente, com parâmetros coincidentes com os atributos DADO do topo e subtopo (ou apenas topo, para operações unárias) da pilha.

Exemplo 4:

```
word X, Y;
byte A, B;
entrada: A shr X and B * Y
```

rotina chamada	código emitido	pilha do atributo DADO (campos INDICE=não, DESLOC=0 e REL=rd)
idvariável		LOC=#A TIPO=byte INDIR=ender
idvariável		LOC=#X TIPO=word INDIR=ender
opbinário	(DLD, A, X)	LOC=% TIPO=byte INDIR=valor
idvariável		LOC=#B TIPO=byte INDIR=ender
idvariável		LOC=#Y TIPO=word INDIR=ender
opbinário	(MUL,B,Y)	LOC=% TIPO=word INDIR=valor
opbinário	(E, %, %)	LOC=% TIPO=word INDIR=valor

III.1.3 - Atribuição

Como já foi visto, consideramos a atribuição como uma operação binária que consiste em armazenar no endereço do primeiro operando, e convertido ao seu tipo, o valor do segundo operando. São três as rotinas semânticas responsáveis pela geração do código intermediário necessário para efetuar tal operação: atribuição comando, atribuição expressão e atribuição for. De acordo com a GTLCA, apenas às duas últimas corresponde um atributo DADO, associado ao resultado, que para atribuição expressão é o valor do operando e no caso de atribuição for é o endereço do primeiro operando. Assim, considerando-se ainda que a instrução gerada por atribuição expressão deixa o valor do segundo operando convertido ao tipo do primeiro no topo da pilha de operandos, o atributo DADO resultante terá LOC=%, INDIR=valor, REL=--, DESLOC=--, INDICE=não e TIPO igual ao do primeiro operando. A rotina atribuição for será estudada em conjunto com as demais rotinas relativas ao comando for.

Exemplo 5:

```
byte B; word W;
entrada: B:= W:=0;
```

rotina chamada	código emitido	pilha do atributo DADO (campos INDICE=não, DESLOC=0 e REL=rd)
idvariável	---	LOC=#B TIPO=byte INDIR=ender
idvariável	---	LOC=#W TIPO=word INDIR=ender
constante	---	LOC=0 TIPO=byte INDIR=valor
atribuição expressão	(ARM,W,0)	LOC=% TIPO=word INDIR=valor
atribuição comando	(ARMC,B,%)	---

III.2 - Comandos de Controle do Fluxo de Execução

Nesta seção será estudado o conjunto de rotinas semânticas necessário à geração de código para comandos de controle do fluxo de execução. Para cada um destes comandos será examinada a sequência de rotinas semânticas que corresponde à sua tradução, bem como a manipulação de atributos e a geração de código intermediário por estas rotinas. Para tanto apresentaremos as produções e as regras de definição de atributos da GTLCA relacionadas, seguidas de uma explicação sobre a ação das rotinas semânticas e de um exemplo, onde as chamadas a rotinas semânticas, o estado das pilhas de atributos envolvidas e o código intermediário emitido serão especificados a cada chamada a rotina semântica.

III.2.1 - Comando Condicional e Expressão Condicional

Sintaxe de Tradução:

COMANDO.CONDICIONAL ::= IF.THEN COMANDO then
| IF.THEN.COM.ELSE COMANDO else

IF.THEN.COM.ELSE ::= IF.THEN COMANDO then else !

IF.THEN ::= IF EXPRESSÃO then if while !

IF := if

EXPRESSÃO.CONDICIONAL ::= IF.THEN.EX.ELSE EXPRESSÃO else expr !

IF.THEN.EX.ELSE ::= IF.THEN EXPRESSÃO then expr !

if while ::= PROG

then else ::= PROG

then expr ::= DADO, PROG

else expr ::= DADO

A rotina if while deve gerar um código tal que as instruções correspondentes ao COMANDO ou EXPRESSÃO seguintes sejam executadas apenas se o resultado da EXPRESSÃO anterior tiver o valor "verdadeiro". Emite portanto código para calcular o valor lógico do resultado da EXPRESSÃO, descrito pelo topo da pilha do atributo DADO, e um desvio-se-falso com o próximo número de referência à frente disponível. Este valor é guardado no topo da pilha do atributo PROG com REL="indef", para ser associado por uma das rotinas then, then else ou then expr ao endereço seguinte ao COMANDO ou EXPRESSÃO que se deseja executar condicionalmente. Da mesma forma as rotinas then else e then expr produzem o código de desvio sobre o COMANDO ou EXPRESSÃO seguintes utilizando o próximo número de referência à frente disponível. Produzem também um atributo PROG com este valor e REL="indef", que será associado por else ou else expr ao endereço seguinte ao COMANDO CONDICIONAL ou EXPRESSÃO CONDICIONAL, respectivamente.

As rotinas then expr e else expr geram também código para colocar o topo da pilha do atributo DADO no topo da pilha de operandos. Como apenas uma dentre estas será executada, ao fim da EXPRESSÃO CONDICIONAL o resultado de uma das duas expressões estará no topo da pilha de operandos. Estes resultados, portanto, devem ter o mesmo tipo, que será "word" se pelo menos um dos dois resultados o for. Esta compatibilização é feita pela rotina else expr, que para tanto consulta o tipo do atributo DADO produzido pela rotina then expr para este fim.

Exemplo 6:

byte A, B, MAX;
Entrada: if A > B then MAX:=A else MAX:=B;

ROTINA SEMÂNTICA CHAMADA	CÓDIGO EMITIDO	PILHA PROG	PILHA DADO	OUTRAS AÇÕES SEMÂNTICAS
idvariável			A	
idvariável			A,B	
opbinário	(MAI,A,B)		%	
if while	(VL,%,-) (DVF,N,-)	(N1,indef)	--	N:=N+1
idvariável			MAX	
idvariável			MAX,A	
atribuição comando	(ARMC,MAX,A)		--	
then else	(DV,N2,-) (DEF,N1,-)	(N2,indef)		N:=N+1
idvariável			MAX	
idvariável			MAX,B	

ROTINA SEMÂNTICA CHAMADA	CÓDIGO EMITIDO	PILHA PROG	PILHA DADO	OUTRAS AÇÕES SEMÂNTICAS
atribuição comando	(ARMC,MAX, B)		--	
else	(DEF,N2,-)	--		

Exemplo 7:

Entrada: MAX:= if A > B then A else B;

ROTINA SEMÂNTICA CHAMADA	CÓDIGO EMITIDO	PILHA PROG	PILHA DADO	OUTRAS SEMÂNTICAS
idvariável			MAX	
idvariável			MAX , A	
idvariável			MAX ,A ,B	
opbinário	(MAI , A , B)		MAX , %	
if while	(VL ,% ,-) (DV ,N1 ,-)	(N1 , indef)	MAX	N:=N+1
idvariável			MAX , A	
then expr	(CG ,A ,-) (DV ,N2 ,-) (DEF ,N1 ,-)	(N2 , indef)	MAX , %	N:=N+1
idvariável			MAX ,% ,B	
else expr	(CG ,B ,-) (DEF ,N2 ,-)	-- --	 MAX	
atribuição comando	(ARMC ,MAX , %)	-- --	-- --	

Observação: Se, no exemplo anterior, tivéssemos

byte B; word A;

o código emitido por expr else seria

(CG, B, -) (EXPS, %, -).

Se, ao contrário, tivessemos

byte A; word B;

o código emitido seria

(CG, B, -) (DV, N3, -) (DEF, N2, -) (EXPS, %, -) (DEF, N3, -)
além da ação semântica N:=N+1.

III.2.2 - Comandos Iterativos

Sintaxe de Tradução:

COMANDO.WHILE ::= WHILE.DO COMANDO fim while

WHILE.DO ::= WHILE ! EXPRESSÃO do if while !

WHILE ::= while $ rótulo !

$ rótulo ::= PROG

A rotina $ rótulo faz com que o endereço de programa relativo ao início da EXPRESSÃO seja guardado na pilha do atributo PROG. A rotina if while, como no COMANDO.CONDICIONAL, gera o cálculo do valor lógico da EXPRESSÃO e um desvio-se-falso para o endereço seguinte ao COMANDO.WHILE. A rotina fim while gera um desvio condicional para o início da EXPRESSÃO (encontrada no subtopo da pilha do atributo PROG), fechando a iteração, e define o valor do topo da pilha do atributo PROG.

Sintaxe de Tradução:

COMANDO.REPEAT ::= REPEAT COMANDO goto
| REPEAT.UNTIL EXPRESSÃO until

REPEAT.UNTIL ::= REPEAT ! COMANDO until

REPEAT ::= repeat $ rótulo !

Neste caso é o endereço inicial do COMANDO o valor do atributo PROG da rotina $ rótulo, para o qual é gerado um desvio incondicional pela rotina goto, ou de acordo com o resultado da EXPRESSÃO pela rotina until, que também gera código para determinar o valor lógico da EXPRESSÃO.

Sintaxe de Tradução:

COMANDO.FOR ::= FOR.STEP.UNTIL.DO COMANDO fim_for

FOR.STEP.UNTIL.DO ::= FOR.STEP.UNTIL ! EXPRESSÃO do limite !
| FOR.STEP ! EXPRESSÃO do limite !

FOR.STEP.UNTIL ::= FOR.STEP ! EXPRESSÃO until step !

FOR.STEP ::=
| FOR.ATRIB EXPRESSÃO step atribuição_for !
| FOR.ATRIB EXPRESSÃO to atribuição_for ! 1 constante !
| FOR.ATRIB EXPRESSÃO downto atribuição_for ! -1 constante !

FOR.ATRIB ::= FOR VARIÁVEL ::=

FOR ::= for

Chamaremos de EXPRESSÃO 1 aquela de segue o sinal de atribuição, de EXPRESSÃO 2 aquela de precede o "until" e de EXPRESSÃO 3 a que precede o "do". A VARIÁVEL, que denominaremos variável de controle, é inicializada com o valor da EXPRESSÃO 1. A EXPRESSÃO 3 fornece o limite para o valor da variável de controle, sendo o COMANDO executado enquanto este limite não for ultrapassado. O resultado da EXPRESSÃO 2 é um valor constante (passo), a ser somado à variável de controle à cada iteração, ou este valor será igual a 1 ou -1 para as opções "to e "downto" respectivamente. A disposição destas ações na memória é a seguinte (o cálculo do passo não provoca emissão de código por ser uma expresão composta de valores constantes):

inicialização da variável de controle;
cálculo do limite;
desvio para TESTE;
LAÇO:
comando: soma do passo à variável de controle;
TESTE:

desvio para LAÇO se variável de controle não ultrapassou limite;

A rotina atribuição for gera código para armazenar o resultado da EXPRESSÃO 1 na variável de controle, e seu atributo DADO guarda o endereço desta. Já os atributos DADO das rotinas step, 1 constante e -1 constante guardam o valor do passo. A rotina limite tem três atributos:
DADO é o endereço onde foi armazenado o valor limite, o primeiro PROG é o número de referência à frente usado pelo desvio para TESTE, e o segundo PROG é o endereço do COMANDO. Além da determinação destes atributos, a rotina gera o armazenamento da EXPRESSÃO 3 numa posição da área de dados que reserva para o limite, e também o desvio para TESTE. A rotina fim for utiliza os cinco atributos anteriores para gerar o código correspondente à soma do passo à variável de controle, à comparação do valor resultante com o limite e a um desvio para nova execução do COMANDO caso a variável de controle seja menor ou igual ao limite (passo positivo) ou maior ou igual ao limite (passo negativo). A variável de controle deve ser simples e não-referida e seu tipo determina os tipos de todas as outras expressões e comparações.

Exemplo 8:

```
word A, B, C;
word(10) V;

Entrada: repeat
              while C do
                   for B:=0 to 9 do
                       V(B):=A;
```

ROTINA SEMÂNTICA	CÓDIGO EMITIDO	PILHA PROG	PILHA DADO	OUTRAS AÇÕES SEMÂNTICAS
$ rótulo		($1,rp)		
$ rótulo		($1,rp)($2,rp)		
idvariável			C	
if while	(VL, C,-) (DVF,N1,-)	($1,rp)($2,rp) (N1,indef)	--	N:=N+1
idvariável			B	
constante			B, 0	
atribuição for	(ARMC,B,0)		B	
1 constante			B,1	

ROTINA SEMÂNTICA	CÓDIGO EMITIDO	PILHA PROG	PILHA DADO	OUTRAS AÇÕES SEMÂNTICAS
constante			B,1,9	
limite	(ARMC,£1,9) (VAG,2,-) (DV,N2,-)	($1,rp)($2,rp) (N1,indef) (N2,indef) ($3,rp)	B,1,£1	N:=N+1
idvariável			B,1,£1,V	
idvariável			B,1,£1,B	
índice			B,1,£1,V(B)	
idvariável			B,1,£,V(B),A	
atribuição comando	(ARMC,V(B),A)		B,1,£1	
fim for	(ADI,B,1) (DEF,N2,-) (MAG,B,£1) (VL,%,-) (DVF,$3,-)	($1,rp)($2,rp) (N1,indef)	--	
fim while	(DV,$2,-) (DEF,N1,-)	($1,rp)		
goto	(DV,$1,-)	--		

III.2.3 - Comando "Case"

Sintaxe de Tradução:

COMANDO.CASE::= CASE.BEGIN.END

CASE.BEGIN.END::= CASE.BEGIN SEQUÊNCIA.COMANDOS end fim case

CASE.BEGIN::= CASE.OF ! begin

CASE.OF::= CASE EXPRESSÃO of case !

CASE::= case

SEQUÊNCIA.COMANDOS::= COMANDO fim caso !
| SEQUÊNCIA.COMANDOS.PTVIRG COMANDO fim caso !

SEQUÊNCIA.COMANDOS.PTVIRG::= SEQUÊNCIA COMANDOS ! ;

case::= PROG, PROG

fim caso::= PROG

A disposição do código do COMANDO.CASE na memória é a seguinte:

```
cálculo da EXPRESSÃO;
desvio para o valor contido em (VETCASOS + EXPRESSÃO);
COMANDO 1;
desvio para o endereço seguinte ao COMANDO.CASE;
.
.
.
.
COMANDO n;
desvio para o endereço seguinte ao COMANDO.CASE;
VETCASOS:
endereço do COMANDO 1
.
.
.
endereço do COMANDO n
```

A rotina case gera o desvio para o caso correspondente ao resultado da EXPRESSÃO, utilizando o próximo número de referência à frente disponível. Reserva ainda o número seguinte a este, que a rotina fim caso usa para gerar um desvio para o endereço seguinte ao COMANDO.CASE cada vez que for chamada. O primeiro número de referência à frente é guardado na pilha do atributo PROG, e além deste a rotina case produz outro atributo PROG com o endereço seguinte ao desvio mencionado, que corresponde ao início do COMANDO 1.

Cada rotina fim caso, por sua vez, gera um atributo PROG com o endereço inicial do COMANDO seguinte, e estes vão sendo acrescentados à pilha PROG. A rotina fim case encontra na pilha PROG os endereços iniciais de todos os casos e a seguir o número a ser associado ao endereço do vetor de casos, que se distingue dos demais por ter o campo REL igual a "indef". Define então este número, gera o vetor com os endereços dos comandos e finalmente associa o próximo endereço de programa ao número seguinte ao associado ao vetor de casos.

Exemplo 9:

word X, A, B, C;

Entrada:
```
case X of
begin
  X:=A;
  X:=B;
  X:=C
end;
```

ROTINA SEMÂNTICA CHAMADA	CÓDIGO EMITIDO	PILHA PROG	OUTRAS AÇÕES SEMÂNTICAS
case	(DVI,N1,X)	(N1,indef)($1,rp)	N:=N+2
atribuição comando	(ARMC,X,A)		
fim caso	(DV,N1+1,-)	(N1,indef)($1,rp) ($2,rp)	
atribuição comando	(ARMC,X,B)		
fim caso	(DV,N1+1,-)	(N1,indef)($1,rp) ($2,rp)($3,rp)	
atribuição comando	(ARMC,X, C)		
fim caso	(DV,N1+1,-)	(N1,indef)($1,rp) ($2,rp)($3,rp)($4,rp)	

ROTINA SEMÂNTICA CHAMADA	CÓDIGO EMITIDO	PILHA PROG	OUTRAS AÇÕES SEMÂNTICAS
fim_case	(DEF,N1,-)		
	(DW,$1,-)		
	(DW,$2,-)		
	(DW,$3,-)		
	(DEF,N1+1,-)		

Observações:

1) Omitimos a pilha de DADO e as chamdas a idvariável.

2) O topo da pilha de PROG não é incluído no vetor de endereços.

3) O atributo correspondente ao endereço do vetor de dados é reconhecido por ter o campo REL igual a "indef", enquanto que todos os outros tinham este igual a "rp".

III.2.4 - Comandos de Desvio:

Sintaxe de Tradução:

DESVIO ::= GOTO.RÓTULO goto rótulo
| GOTO.CHAVE goto chave

GOTO.RÓTULO ::= GOTO idrótulo id rótulo !

GOTO.CHAVE ::= GOTO.ID.CHAVE.ABRE EXPRESSÃO) índice !

GOTO.ID.CHAVE.ABRE::= GOTO.ID.CHAVE ! (

GOTO.ID.CHAVE ::= GOTO idchave idchave !

GOTO ::= goto

idrótulo ::= PROG

idchave ::= DADO

índice ::= DADO

A rotina idrótulo produz um atributo PROG com o endereço para o qual a rotina goto rótulo gerará um desvio. A rotina id chave cria um atributo DADO com o endereço do "switch", enquanto que o atributo DADO da rotina índice define o elemento do "switch" para o qual será desviada a execução, pela rotina goto chave.

Exemplo 10:

```
label L1, L2;
switch S=(L1, L2);
```

Entrada: goto L1; goto S(0);

ROTINA SEMÂNTICA CHAMADA	CÓDIGO EMITIDO	PILHA DADO	PILHA PROG
id rótulo			(L1,rp)
goto rótulo	(DV,L1,-)		
id chave		S	
constante		S, 0	
índice		S(0)	
goto chave	(DVD,S(0),-)	--	

Observação:
Supusemos o rótulo L1 já definido. Caso contrário, o atributo PROG gerado pro idrótulo seria (L1,indef).

III.2.5 - Chamada de Procedimento

Sintaxe de Tradução:

```
CHAMADA.FUNÇÃO::= ID.FUNÇÃO chamada função !
                | ID.FUNÇÃO.ABRE.FECHA chamada função !

ID.FUNÇÃO.ABRE.FECHA::=
     ID.FUNÇÃO.ABRE ! LISTA.PARÂMETROS.EFETIVOS )

ID.FUNÇÃO.ABRE::= ID.FUNÇÃO ! (

ID.FUNÇÃO ::= idfunção id função !

id função ::= ROT

chamada função ::= DADO

CHAMADA.ROTINA ::= ID.ROTINA chamada rotina
                 | ID.ROTINA.ABRE.FECHA chamada rotina

ID.ROTINA.ABRE.FECHA::=
     ID.ROTINA.ABRE ! LISTA.PARÂMETROS.EFETIVOS )

ID.ROTINA.ABRE ::= ID.ROTINA ! (

ID.ROTINA ::= idrotina id rotina !

LISTA.PARÂMETROS.EFETIVOS ::= EXPRESSÃO parâmetro
     | LISTA.PARÂMETROS.EFETIVOS.VIRG ::= EXPRESSÃO parâmetro

LISTA.PARÂMETROS.EFETIVOS.VIRG ::= LISTA.PARÂMETROS.EFETIVOS ,

id rotina ::= rot
```

A rotina idrotina cria um atributo ROT com informações sobre a rotina e seus parâmetros. Cada chamada à rotina parâmetro gera o código necessário para que o resultado da EXPRESSÃO anterior ocupe o topo da pilha de operandos, convertido ao tipo do parâmetro formal. Para tanto, idrotina inicializa o campo PARMATUAL de ROT com o valor 1, o qual é incrementado de 1 a cada execução de parâmetro. Finalmente chamada rotina gera uma instrução de chamada à rotina.

Exemplo 11:

```
byte A, B;
procedure P (byte X; word Y);
```

Entrada: P(A, A+B);

ROTINA SEMÂNTICA CHAMADA	CÓDIGO EMITIDO	PILHA DADO	PILHA ROT
idrotina			LOC=#P TIPO=proc REL=rp PARMATUAL=1 NPARMS=2 TIPOPARM=byte,word
idvariável		A	
parâmetro	(CG,A,-)	--	LOC=#P TIPO=proc REL=rp PARMATUAL=2 NPARMS=2 TIPOPARM=byte,word
idvariável		A	
idvariável		A, B	
opbinário	(ADI,A,B)	%	
parâmetro	(EXPS,%,-)	--	LOC=#P TIPO=proc REL=rp PARMATUAL=3 NPARMS=2 TIPOPARM=byte,word
chamada rotina	(CHS,P,-)	--	--

Observações:

1) O processo de chamada de função é essencialmente o mesmo, mas o TIPO do atributo ROT criado por idfunção é transmitido ao atributo DADO gerado pela rotina chamada função.

III.3 - Declaração de Procedimento

Sintaxe de Tradução:

DECLARAÇÃO.ROTINA.INTERNA::= PROC.ID.PTVIRG COMANDO return
| TIPO.PROC.ID.PTVIRG COMANDO return

PROC.ID.PTVIRG::= PROC.ID ! ; recepção
|PROC.ID.ABRE.FECHA ! ; recepção

PROC.ID.ABRE.FECHA::=
PROC.ID.ABRE ! SEQUÊNCIA.ESPECIFICAÇÃO.PARÂMETROS

PROC.ID.ABRE::= PROC.ID ! (

PROC.ID::= PROCEDURE identificador decl rotina !

PROCEDURE::= procedure

TIPO.PROC.ID.PTVIRG::= TIPO.PROC.ID ! ; recepção
|TIPO.PROC.ID.ABRE.FECHA ! ; recepção

TIPO.PROC.ID.ABRE.FECHA::=
TIPO.PROC.ID.ABRE ! SEQUÊNCIA.ESPECIFICAÇÃO.PARÂMETROS)

TIPO.PROC.ID.ABRE::= TIPO.PROC.ID ! (

TIPO.PROC.ID::= TIPO.PROC identificador decl rotina !

```
TIPO.PROC::= TIPO ! procedure

TIPO::= tipo tipo !

SEQUÊNCIA.ESPECIFICAÇÃO.PARÂMETROS::= ESPECIFICAÇÃO.PARÂMETROS
|SEQUÊNCIA.ESPECIFICAÇÃO.PARÂMETROS.PTVIRG
                          ESPECIFICAÇÃO.PARÂMETROS

SEQUÊNCIA.ESPECIFICAÇÃO.PARÂMETROS.PTVIRG::=
SEQUÊNCIA.ESPECIFICAÇÃO.PARÂMETROS ;

ESPECIFICAÇÃO.PARÂMETROS::= TIPO SEQUÊNCIA.PARÂMETROS

SEQUÊNCIA.PARÂMETROS::= IDENTIFICADOR
                      |SEQ.PARMS.VIRG.IDENTIFICADOR

SEQ.PARMS.VIRG.IDENTIFICADOR::=
     SEQ.PARMS.VIRG identificador decl parm

SEQ.PARMS.VIRG::= SEQUÊNCIA.PARÂMETROS ,

IDENTIFICADOR::= identificador decl parm

BLOCO::= BEGIN.END

BEGIN.END::= BEGIN SEQUÊNCIA.COMANDOS end

BEGIN::= begin

SEQUÊNCIA.COMANDOS::= COMANDO
                    | SEQUÊNCIA.COMANDOS.PTVIRG COMANDO

SEQUÊNCIA.COMANDOS.PTVIRG::= SEQUÊNCIA.COMANDOS ;
```

A rotina tipo modifica o valor do atributo TIPO. A rotina decl parm cria um atributo DROT com o valor atual do contador de dados, o tipo da função (se for o caso) obtido em TIPO, e zero parâmetros. Cada chamada a decl parm incrementa o contador de

parâmetros de um, e acrescenta o tipo do parâmetro, obtido em TIPO, à lista de tipos de parâmetros do atributo DROT. A rotina recepção retira os parâmetros encontrados da pilha de operandos e cujo número e tipo são descritos pro ROT, armazenando-os na área de dados para utilização pelo COMANDO da DECLARAÇÃO.DE.ROTINA.INTERNA. Finalmente return gera uma instrução para colocar o resultado da função no topo da pilha de operandos (se for o caso) e outra de retorno de procedimento. A rotina decl rotina abre espaço na área de dados para o resultado da função e cada chamada a decl parm faz o mesmo para o parâmetro correspondente.

Exemplo 12:

Entrada: word procedure FW (byte A, B; word C); return;

ROTINA SEMÂNTICA CHAMADA	CÓDIGO EMITIDO	PILHA ROT	VARIÁVEL TIPO
tipo			word
decl rotina	(VAG,2,-)	LOC=£1 TIPO=word NPARMS=0 TIPOPARM=--	
tipo			byte
decl parâmetro	(VAG,1,-)	LOC=£1 TIPO=word NPARMS=1 TIPOPARM=byte	
decl parâmetro	(VAG,1,-)	LOC=£1 TIPO=word NPARMS=2 TIPOPARM=byte,byte	
tipo			word
decl parâmetro	(VAG,2,-)	LOC=£1 TIPO=word NPARMS=3 TIPOPARM=byte,byte, word	
recepção	(DCG,£1+3,-) (DCG,£1+2,-) (DCG,£1+1,-)	LOC=£1 TIPO=word NPARMS=3 TIPOPARM=byte,byte, word	

ROTINA SEMÂNTICA CHAMADA	CÓDIGO EMITIDO	PILHA ROT	VARIAVEL TIPO
return	(CG,£1,-)	LOC=£1 TIPO=word NPARMS=3 TIPOPARM=byte,byte word	
return	(CG,£1,-) (RET,-,-)		

IV - Tradução do Código Intermediário em Instruções do Processador

No código intermediário emitido pelas rotinas semânticas já estão resolvidas a estrutura do programa objeto e a ordem das operações a serem executadas. A geração do código se completará, portanto, com a transformação deste código intermediário em instruções do computador objeto, a qual pode ser feita sobre a totalidade do programa ou sobre cada instrução intermediária emitida. A primeira opção oferece maiores possibilidades para se gerar um código eficiente, uma vez que o contexto em que cada operação é realizada será conhecido. Nosso estudo , no entanto, será voltado para a geração de código em um só passo, quando cada instrução intermediária é imediatamente traduzida em instruções de máquina, sem preocupação com o contexto em que foi produzida. Desta maneira, após o estabelecimento da correspondência entre a arquitetura auxiliar definida para o código intermediário e a arquitetura do processador, a determinação das instruções de máquina que implementem uma instrução intermediária é quase sempre imediata, exceto no caso das operações binarias (inclusive atribuição), quando existe maior diversidade de situação em que cada operando pode se encontrar e alterações em um operando devem ser feitas levando-se em conta o outro operando, até que se obtenha uma combinação para a qual a operação em questão possa ser realizada.

Nossa preocupação central no presente capítulo será resolver este problema de maneira que tenhamos um código tão eficiente quanto possível. Para tanto apresentaremos um esquema teórico em que se escolhe a melhor sequência de instruções para implementação de uma operação binária. Em seguida veremos sua aplicação ao compilador LPS, onde a eficiência do código gerado prende-se principalmente ao espaço que este ocupará na memória, opção que, não obstante, coincidirá frequentemente com a obtenção do código de menor tempo de execução.

IV.1 - Esquema para Tradução de Operações Binárias

Descreveremos aqui um esquema lógico capaz de representar a situação dos operandos de uma operação binária e indicar as instruções necessárias para realizá-la. A base deste modelo é o conjunto de formas com que o processador acessa um par de operandos unido ao conjunto de operações binárias intermediárias, sendo as instruções consideradas enquanto efetuadoras de transformações entre elementos deste conjunto união. Verificaremos que estas transformações podem ser modeladas pelas arestas de um digrafo onde os elementos do conjunto união acima descrito constituem os nós, e que o problema de escolha da melhor sequência de instruções para realizar uma dada operação binária reduz-se à determinação do caminho mínimo entre dois nós deste grafo. Iniciaremos com um exemplo ilustrativo do problema, para definir em seguida o modelo geral.

Exemplo 1: Considerando como máquina objeto o micro-processador INTEL 8080 reduzido aos registradores A e B e à pilha e sendo dois operandos OP1 e OP2, cada um ocupando um byte de memória em endereço conhecido, determinar:

a) sequências de instruções para calcular OP1+OP2 e OP1-OP2.
Seja o conjunto
G1 = {MM,MA,MB,MT,AM,AB,AT,BM,BA,BT,TM,TA,TB,TS,ST},
onde cada elemento descreve uma possível localização do par de operandos, atingível à partir da situação inicial de ambos na memória. A primeira letra refere-se a OP1 e a segunda a OP2, e cada letra significa:

M - operando na memória;
A - operando no registrador A;
B - operando no registrador B;
T - operando no topo da pilha;
S - operando no sub-topo da pilha;

Sendo soma e subtração as operações que desejamos realizar, estendemos o conjunto anterior a

GO = { MM,MA,MB,MT,AM,AB,AT,BM,BA,BT,TM,TA,TA,TS,ST,ADI,SUB }

onde os elementos acrescentados correspondem às operações de adição (ADI) e subtração (SUB).

Com base no conjunto GO, definimos o digrafo (GO,RO) onde a relação RO é descrita por

x RO y <=> "existe uma instrução do processador capaz de transformar um par de operandos no estado descrito por x para o estado descrito por y".

No caso dos nós ADI e SUB esta transformação significa a realização da operação aritmética correspondente. Na figura 1 está representado o dígrafo (GO,RO). Cada aresta possui dois rótulos, indicados na tabela 1, o primeiro dos quais indica uma das instruções capazes de realizar a transformação e o outro, representa um custo atribuído à transformação - no caso o tamanho da instrução em bytes. Desta forma, o caminho de um nó até o nó ADI fornece, através do primeiro rótulo de cada aresta percorrida, uma sequência de instruções capaz de efetuar a soma de dois operandos que se encontrem inicialmente no estado por ele descrito, o mesmo acontecendo para o nó SUB com relação a operação de subtração.

Partindo portanto do nó MM, o grafo da figura 1 nos fornece, entre outras, as seguintes sequências de instruções para calcular OP1+OP2

```
LDA OP1; MOV B,A; LDA OP2; ADD 3;
e
LDA OP2; MOV B,A; LDA OP1; ADD B;
```

correspondendo respectivamente aos caminhos

```
MM --> AM --> BM --> BA --> ADI
e
MM --> MA --> MB --> AB --> ADI.
```

Para a subtracao temos, por exemplo

LDA OP2; MOV B,A; LDA OP1; SUB B;
e também
LDA OP1; PUSH PSW; LDA OP2; MOV B,A; POP PSW; SUB B;

respectivamente associados aos caminhos

MM --> MA --> MB --> AB --> SUB
e
MM --> AM --> TM --> TA --> TB --> AB --> SUB.

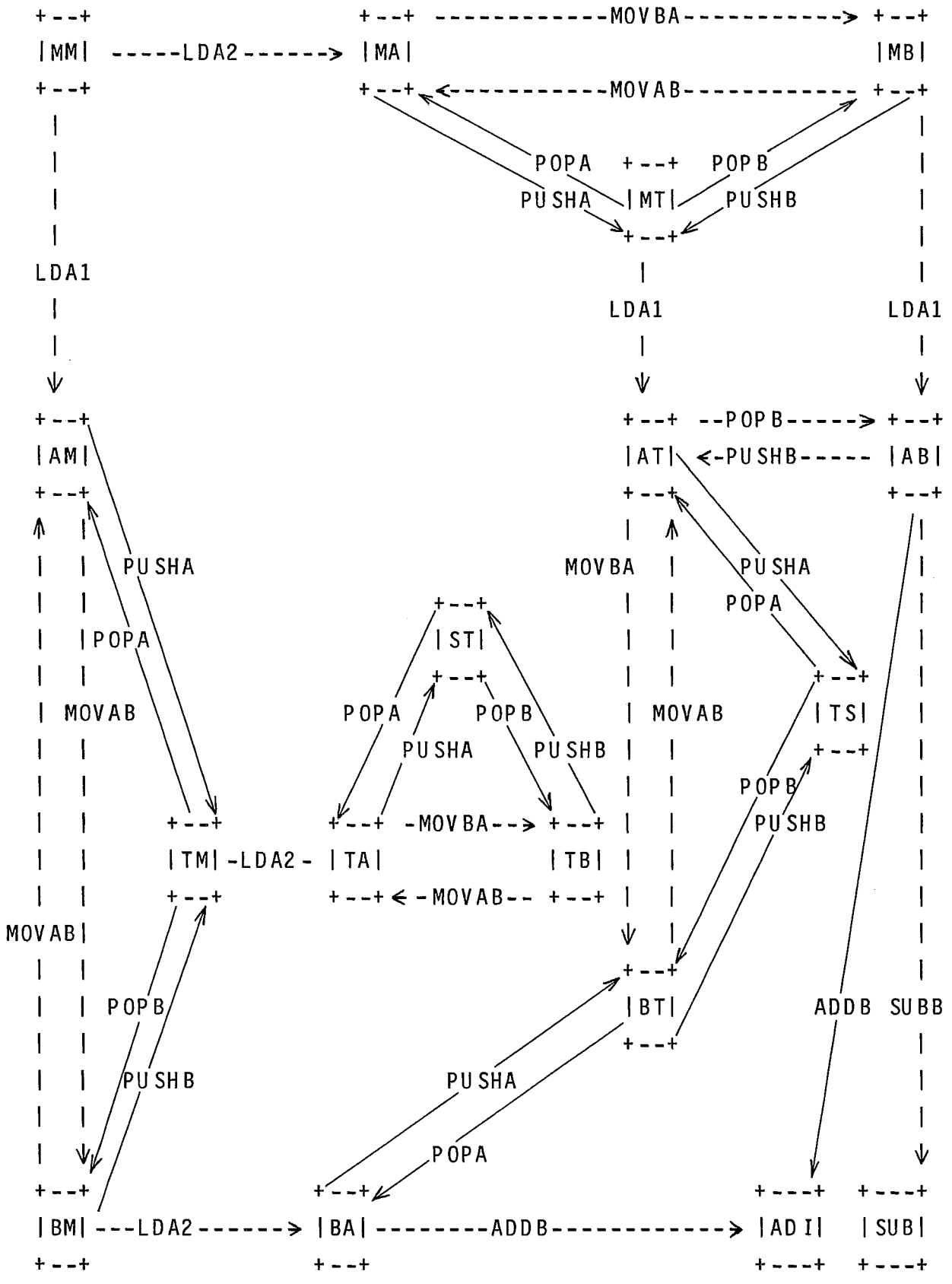

MM
LDA2
MA
MOVBA
MB
MOVAB
POPA
PUSHA
MT
POPB
PUSHB
LDA1
LDA1
LDA1
AM
POPB
AT
PUSHB
AB
PUSHA
MOVBA
PUSHA
POPA
ST
POPA
POPB
MOVAB
TS
PUSHA
PUSHB
POPB
PUSHB
TM
LDA2
MOVBA
TA
TB
MOVAB
MOVAB
POPB
PUSHB
BT
ADDB
SUBB
PUSHA
POPA
BM
LDA2
BA
ADDB
ADI
SUB

Figura 1

Representação	Instrução	Custo
LDA1	LDA OP1	3
LDA2	LDA OP2	3
MOVBA	MOV B,A	1
MOVAB	MOV A,B	1
PUSHA	PUSH PSW	1
PUSHB	PUSH BC	1
POPA	POP PSW	1
POPB	POP BC	1
ADDB	ADD B	1
SUBB	SUB B	1

Tabela 1

b) as sequências de instruções que realizem OP1+OP2 e OP1-OP2 com o menor gasto possível de memória.

No grafo da figura 1, o segundo rótulo de uma aresta fornece o número de bytes ocupados em memória pela instrução dada pelo primeiro rótulo. Somando portanto os valores do segundo rótulo das arestas percorridas temos o custo do caminho que é o espaço ocupado em memória pela sequência correspondente. Para os caminhos determinados no ítem anterior temos, por exemplo, os seguintes custos:

MM -> AM -> BM -> BA -> ADI, custo 8 ; (1)

MM -> MA -> MB -> AB -> ADI, custo 8 ; (2)

MM -> MA -> MB -> AB -> SUB, custo 8 ; e (3)

MM -> AM -> TM -> TA -> TB -> AB -> SUB, custo 10. (4)

A menor sequência de instruções capaz de calcular OP1+OP2 com OP1 e OP2, inicialmente na memória, será aquela formada pelos primeiros rótulos das arestas do caminho de custo mínimo que

comece no nó MM e termine no nó ADI, isto é, do caminho tal que a soma dos segundos rôtulos de suas arestas seja mínima. O mesmo se aplica ao cálculo de OP1-OP2, quando o caminho mínimo a ser determinado deve começar no nó MM e terminar no nó SUB. Por observação do grafo da figura 1, examinando os possíveis caminhos entre os nós, verificamos que oss caminhos (1) e (2) acima são os de custo mínimo de MM para ADI e que o caminho (3) é o de menor custo de MM para SUB. Mais adiante será estudado um algoritmo para resolver eficientemente este problema.

Definição: Esquema para Tradução de Operações Binárias

Seja o conjunto finito A de acessos a dados. Seus elementos estão associados às diversas maneiras do processador tratar um dado de cada um dos tipos manipulados pela linguagem, por exemplo: um byte ou uma palavra num endereço de memória direta ou indiretamente conhecido, byte ou palavra em registrador ou na memória apontada por registrador. Em suma, os elementos de A descrevem as possíveis localizações físicas de cada tipo de operando, bem como as diversas formas se segundo as quais o processador efetua sua busca e utilização, isto é, o acesso ao operando.

Seja o conjunto

$E = \{(x,y) \in A \times A \mid$ dois operandos podem ocupar as localizações descritas por x e y simultâneamente$\}$.

Aos elementos de E denominaremos estados, e seu sentido é informar o estado em que se encontra um par de operandos com relação ao acesso pelo processador.

Seja o conjunto finito B de operações binárias intermediárias tipificadas. Seus elementos representam as operações intermediárias para as quais desejamos gerar código no processador em questão. O tipo de cada operação deve estar determinado pelo elemento correspondente do conjunto B, se isto for semanticamente relevante na linguagem que está sendo traduzida. Isto é, uma mesma operação intermediária pode originar mais de um elemento

no conjunto B se a linguagem admitir operações com tipos diferentes.

Sobre o conjunto (E u B) definimos uma relação R tal que, sendo x e y elementos de (E u B),

x R y <==> "existe uma instrução do processador que transforma um par de operandos descrito por x num par de operandos descrito por y".

Cabe aqui a seguinte observação: muitas vezes o processador em questão não dispõe de instruções capazes de efetuar todas as operações binárias da linguagem para a qual o código está sendo gerado. O procedimento normalmente adotado é o de definir um conjunto de rotinas, que denominaremos rotinas intrínsecas, que recebem os operandos em localização bem definida, por exemplo, em dois registradores determinados, e retornam o resultado da operação também em localização dem definida. Neste caso, tudo se passa como se a instrução de chamada de cada uma destas rotinas seja a instrução que implementa a operação correspondente, com os operandos no estado requerido pela entrada da rotina.

Seja ainda o conjunto I de todas as instruções do processador.

O esquema para tradução de operações binárias (ETOB) é um digrafo G=(E u B,R), sobre as arestas do qual definimos dois rótulos f e g, sendo:

g:R->|R+ uma função que associa a cada aresta (x,y) o menor gasto possível para se realizar a transformação do estado x para o estado y, segundo algum critério pré-definido. Este critério pode ser o espaço ocupado pela instrução, o tempo gasto na sua execução ou uma conjunção destes dois fatores. Denominaremos g(x,y) de custo de aresta (x,y).

f:R-> I uma função tal que f(x,y) é a instrução capaz de realizar a transformação de x para y com o menor gasto possível. No caso de haver mais de uma instrução em tais condições, escolhemos qualquer uma delas para fazer parte do esquema.

Assim sendo, a melhor sequência de instruções para efetuar uma operação binária b entre dois operandos no estado a é dada por:

f(a0,a1) ; f(a1,a2) ; ... ; f(an-1,an) onde a0=a, an=b e

g(a0,a1) + g(a1,a2) +... + g(an-1,an) é mínimo.

Isto é, a sequência de instruções correspondente ao caminho de custo mínimo entre os nós a e **b**. A determinacao do caminho de custo mínimo, ou simplesmente caminho mínimo, é um problema já bastante estudado na teoria dos grafos. O algoritmo para sua solução, que apresentamos a seguir, é uma variação do algoritmo de Dijkstra (DE0,5).

Algoritmo **1**: Determinação de um caminho de custo mínimo entre dois nós de um digrafo.
O algoritmo funciona atribuindo dois rótulos aos nós do digrafo. O primeiro, c:(E u B) -> R+, mede o custo do caminho deste nó até o nó final. Pode assumir no decorrer da execução do algoritmo diversos valores temporários para um mesmo nó, até que receba um valor permanente, que será o custo do caminho mínimo deste nó até o nó final. O segundo rótulo, s:(E u B)->(E u B), assumirá da mesma forma diversos valores temporários para um mesmo nó, mas seu valor permanente indicará o sucessor deste nó no caminho mínimo até o nó final. Começamos atribuindo o valor temporário "infinito" ao rótulo c de todos os nós do grafo, exceto o nó final, para o qual o rótulo c recebe o valor permanente zero. A cada iteração, outro nó recebe um valor permanente para os rótulos c e s, segundo as seguintes regras:

1. Cada nó i que ainda não tenha recebido um rótulo c permanente, recebe novo valor temporário para o rótulo **c** dado por

c(i) = mínimo antigo valor c(i), c(j) + g(i,j)

onde j é o nó cujos rótulos receberam valor permanente na itera-

ção anterior. No caso de não existir a aresta (i,j) supomos g(i,j)="infinito".

2. Se o valor mínimo escolhido em 1 para c(i) é a soma

c(j) + g(i,j)

então s(i) recebe o valor **j**.

3. É escolhido o menor valor do rótulo **c** entre todos os temporários, que passa a ser o valor permanente para aquele nó. Para o mesmo nó o valor do rótulo s torna-se também permanente. Em caso de empate, qualquer dos candidatos pode ser escolhido.

Os três passos são repetidos ordenadamente, até que:

1. o nó inicial receba rótulos **c** e **s** permanentes, ou

2. no momento da determinação do próximo nó a receber rótulos permanentes, todos os valores temporários dos rótulos dos nós sejam iguais a "infinito". No primeiro caso os rótulos **s**, a partir do nó inicial, indicam o caminho mínimo. No segundo caso não existe nenhum caminho do nó inicial até o nó final.

O primeiro nó rotulado permanentemente está a uma distância zero do nó final. O segundo nó a receber valores permanentes para os rótulos c e s é o nó mais próximo ao final. Na outra iteração é rotulado permanentemente o segundo nó mais próximo ao final, e assim por diante. O funcionamento do algoritmo baseia-se ainda no fato de que, se um caminho mínimo de um nó a para um nó **b** passa pelo nó d, então o trecho deste caminho entre os nós **d** e b é um caminho mínimo de d até **b**.

Assim sendo, a função s fornece, ao término do algoritmo, não só um caminho mínimo do nó inicial a até o nó final **b**, como também caminhos mínimos para todos os nós por onde passe o caminho de a até b. Da mesma forma, se eliminarmos a primeira condição de parada do algoritmo, a função s fornecerá caminhos mínimos de todos os nós até o nó final, para os quais algum caminho exista. Se, por outro lado, modificarmos a primeira condição de

parada para interromper a execução quando todo um conjunto de nós especificado tenha recebido rótulos permanentes, a função s indicará caminhos mínimos de todos os elementos deste conjunto até o nó final.

Conjuntos de Nós Iniciais e Finais: Definido um esquema como o acima descrito, o compilador é capaz de escolher a sequência adequada de instruções para cada ocorrência de geração binária, bastando para isso conhecer o digrafo com seus rótulos **f** e **g**. Isto acarretaria,no entanto, um grande aumento no tempo de compilação, o que torna mais atraente a opção de termos já determinados, em tempo de construção do compilador, todos os caminhos de interesse para a compilação, ou seja, todos os caminhos mínimos entre todos os pares que podem ocorrer como iniciais e finais durante a compilação de qualquer programa.

O conjunto de nós finais coincide com o conjunto B de operações binárias intermediárias tipificadas. Já o conjunto de nós iniciais (ou estados iniciais, pois apenas os elementos de AxA podem ser nós iniciais) depende das formas de acesso em que podem se encontrar os operandos das instruções intermediárias, isto é, do conjunto de representações do atributo DADO descritor dos operandos (vide 3.1.1) e da implementação da pilha de operandos, onde estão os resultados intermediários. Além disso, o conjunto de estados iniciais depende do estado final que se quer alcançar, ou mais precisamente, dependem ambos do tipo dos operandos envolvidos. Considerando o esquema para geração de operações binárias da figura 1, observamos que o resultado de uma operação binária é sempre dado no registrador A. No momento em que iniciamos a emisssão das instruções para esta operação, o registrador A pode estar ocupado pelo resultado de outra operação da mesma expressão, caso em que é necessário liberá-lo, o que pode ser feito movendo-se seu conteúdo para a pilha da UCP. Isto equivale a dizer que a pilha de operandos é implementada pela pilha de UCP e pelo registrador A, onde está o operando do topo. O conjunto de nós iniciais para aquele exemplo seria, portanto: MM, MA, AM, TA , e apenas a partir destes é necessário determinar os caminhos mínimos até os nós finais ADI e SUB .

Neste caso, o conjunto de nós iniciais independe do nó final objetivo.

Exemplo 2: Para o esquema definido no exemplo 1, determinar os caminhos mínimos do conjunto de nós iniciais até o conjunto de nós finais.

nós iniciais: MM, MA, AM, TA
nós finais: ADI, SUB

É necessário executar o algoritmo 1 duas vezes, uma para cada nó final. Representaremos os rótulos c e s através de uma tabela, de tamanho igual ao número de nós, onde os valores permanentes serão os últimos da coluna. Os elementos da coluna a são da forma c(a) s(a) ou estão em branco se c(a)="infinito" ou se a já recebeu rótulo permanente numa iteração anterior. Cada linha representa uma iteração no algoritmo.

O primeiro nó inicial a receber o rótulo permanente s.ADI foi TA, na quinta iteração. Quando a condição de término foi atingida, tanto para s.ADI quanto para s.SUB, todos os nós haviam recebido rótulo permanente, exceto o correspondente à outra operação, pois entre as duas operações não existe nenhum caminho. O nó MM foi o último a receber rótulo permanente, pois é o mais distante tanto de ADI quanto de SUB.

MM	MA	MB	MT	AM	AB	AT	BM	BA	BT	TM	TA	TB	TS	ST
					1ADI			1ADI						
		4AB				2AB		1ADI				2AB		
		4AB				2AB	4BA		2BA		2BA	2AB		
		4AB	5AT				4BA		2BA		2BA	2AB	3AT	
		4AB	5AT				4BA				2BA	2AB	3AT	
		4AB	5AT				4BA			5TA		2AB	3AT	3TA
		4AB	5AT				4BA			5TA			3AT	3TA
		4AB	5AT				4BA			5TA				3TA
		4AB	5AT				4BA			5TA				
	5MB		5AT				4BA			5TA				
	5MB		5AT	5BM						5TA				
8MA			5AT	5BM						5TA				
8MA				5BM						5TA				
8MA										5TA				
8MA														

Tabela 2 - Determinação dos caminhos mínimos até ADI

MM	MA	MB	MT	AM	AB	AT	BM	BA	BT	TM	TA	TB	TS	ST
					1 SUB									
						2AB						2AB		
		4AB							3AT			2AB	3AT	
		4AB	5AT						3AT		3TB		3AT	3TB
		4AB	5AT					4BT			3TB		3AT	3TB
		4AB	5AT					4BT		6TA			3AT	3TB
		4AB	5AT					4BT		6TA				3TB
		4AB	5AT					4BT		6TA				
	5MB		5AT					4BT		6TA				
	5MB		5AT				7BA			6TA				
8MA			5AT				7BA			6TA				
8MA							7BA			6TA				
8MA				7TM			7BA							
8MA							7BA							
8MA														

Tabela 3 - Determinação dos caminhos mínimos até SUB

Nó	s.ADI(Nó)
MM	MA
MA	MB
MB	AB
AM	BM
AB	ADI
BM	BA
BA	ADI
TA	BA

Tabela 4 - S.ADI

Nó	s.SUB(Nó)
MM	MA
MA	MB
MB	AB
AM	TM
TM	TA
TA	TB
TB	AB
AB	SUB

Tabela 5 - S.SUB

Os caminhos desejados estão nas tabelas 4 e 5, onde cada linha indica o sucessor do nó no caminho escolhido até ADI ou SUB, respectivamente. Verificamos a viabilidade da representação dos diversos caminhos até um mesmo nó numa mesma linha pois, por exemplo, o caminho de MM até ADI segue por MA, e coincide a partir daí com o caminho de MA até ADI. Verificamos também que boa parte dos nós não faz parte de nenhum caminho, e não precisa ser representada.

IV.2 - Esquema para Tradução de Operações Binárias no LPS/300

Descrevemos nesta seção a aplicação do esquema para tradução de operações binárias ao compilador LPS/300, cuja máquina-objeto é o COBRA-300, dotado de micro-processador INTEL-8080. Serão apresentadas as implementações da pilha de operandos, do digrafo conforme definido em 4.1, e do resultado da aplicação do algoritmo 1 a este digrafo. Na ocasião, serão vistas algumas simplificações feitas com o intuito de diminuir o tamanho e o número das tabelas resultantes, e concomitantemente o número de execuções do algoritmo necessárias para produzir tais tabelas. Estas simplificações são possíveis devido a características próprias do processador INTEL-8080, mas aplicam-se aproximadamente da mesma forma a muitos outros processadores, que satisfaçam às condições necessárias.

A primeira providência é estabelecer a correspondência entre a arquitetura auxiliar do código intermediário e a arquitetura do processador 8080. A pilha de ligação é implementada pela pilha do processador, já que as instruções "CALL" e "RET" operam sobre esta pilha. O registrador de estado corresponde ao bit Z da palavra de estado (PSW) do processador, sendo o valor "verdadeiro" associado ao estado "ligado" e o valor "falso" ao estado "desligado". A pilha de operandos é o que veremos a seguir.

Pilha de Operandos: O processador 8080 admite dados inteiros de oito bits (bytes) e dezesseis bits (word). Para operar inteiros de oito bits, usa como acumulador o registrador A, e para inteiros de dezesseis bits o registrador HL. Assim, o resultado de uma operação entre operandos do primeiro tipo ocupará o registrador A e do segundo o registrador HL. O conjunto de instruções do 8080 entretanto, não é suficiente para realizar todas as operações binárias da linguagem LPS, tendo sido necessário um conjunto de rotinas, denominadas intrînsecas, que preenchesse as lacunas do jogo de intruções, efetuando operações como por exemplo as de multiplicação e divisão entre operandos do tipo byte, e todas as operações, exceto a soma, entre operandos do tipo word. Mantendo a compatibilidade com as demais intruções, foram escolhidos os registradores A e HL para conterem os resultados, do tipo byte e word respectivamente, retornados pelas rotinas intrînsecas.

Para que, durante a obtenção da sequência de intruções que efetuem uma operação binária, não seja necessário preocupar-se com a ocupação de registradores que não os relacionados aos próprios operandos, foi resolvido que resultados parciais anteriores seriam transferidos para a pilha do processador. No caso das operações unárias, porém, apenas o registrador correspondente ao tipo do operando é liberado. Desta forma, a pilha de operandos é constituída pelos registradores A, HL e a pilha do processador, organizados da seguinte forma:

- o topo da pilha de operandos ocupa sempre um dos registradores A ou HL;
- o subtopo da pilha de operandos pode ocupar o outro destes registradores ou o topo da pilha do processador; - os demais operandos estão na pilha do processador.

A figura abaixo ilustra as possíveis configurações da pilha de operandos:

Figura 2 - Configurações da Implementação da Pilha de Operandos

Estados Iniciais: Conforme foi visto em 3.1, a descrição de um operando pelo atributo DADO pode ou não referir-se à pilha de operandos. Estes últimos descrevem constantes, variáveis, variáveis referidas e variáveis referidas ou não com índice variável referido não. Nos dois últimos casos, em virtude das características do jogo de instruções do processador 8080, o acesso ao valor do operando passa obrigatoriamente pelo cálculo de seu endereço, que se dará no registrador HL. Portanto, é possível representar todas as formas de acesso a variáveis referidas ou indexadas pelo mesmo símbolo, e assim temos os seguintes acessos a operandos que não envolvem a pilha de operandos:

RI1 - variáveis byte referidas ou indexadas
RI2 - variáveis word referidas ou indexadas
E1 - variáveis byte de endereço conhecido
E2 - variáveis word de endereço conhecido
V1 - constantes byte
V2 - constantes word

Os endereços das variáveis considerados conhecidos podem ser absolutos, relocáveis ou externos. O valor das constantes byte é sempre absoluto, mas o das constantes word pode ser absoluto ou relocável. Constantes de valor relocável, porém, terão seu valor previamente carregado na pilha de operandos, e não serão representados por "V2".

Além dos acima, temos os operandos cujo valor se encontra na pilha de operandos, aqueles cujo endereço está na pilha de operandos e ainda as variáveis indexadas com índice gerado na pilha de operandos. Estas últimas serão previamente transformadas em variáveis com endereço na pilha de operandos, reduzindo aos dois primeiros os casos de interesse. Em virtude da forma como foi implementada a pilha de operandos, os seguintes acessos podem ocorrer:

VA - valor do operando no registrador A
VHL - valor do operando no registrador HL
VP1 - valor do operando byte no topo da pilha
VP2 - valor do operando word no topo da pilha
EHL1 - endereço do operando byte no registrador HL
EHL2 - endereço do operando word no registrador HL
EP1 - endereço do operando byte no topo da pilha
EP2 - endereço do operando word no topo da pilha

Estes são portanto os elementos do conjunto de acessos que descrevem os operandos iniciais. Chamando de AI este conjunto teremos

$AI = AIM \cup AIP \cup AIR$
$AIM = \{RI1, RI2, E1, E2, V1, V2\}$
$AIP = \{VP1, VP2, EP1, EP2\}$

AIR = {VA, VHL, EHL1, EHL2}

Para determinar o conjunto EI de estados iniciais, observemos primeiramente que, como um registrador só pode estar alocado a um operando, não existirão os estados em que ambos os acessos refiram-se ao mesmo registrador.

Além disso, pelas configurações possíveis para a implementação da pilha de operandos, apenas o operando esquerdo pode ter seu valor ou endereço na pilha do processador, e apenas se o operando direito tiver seu valor ou endereço em registrador, isto é

$$EI = \{(x,y)\ AI \times AI \mid (x \text{ e } y \text{ não se referem ao mesmo registrador}) \text{ e } (y \notin AIP) \text{ e } (x \in AIP \text{ então } y \in AIR)$$

São estados iniciais, por exemplo:

(E1,E1), (RI1,V2), (VA,RI1), (E2,VHL), (EP1,VHL), (VP1,VA).

Não são estados iniciais:

(VA,VA), (VHL,EHL2), (VA,VP1), (EP2,RI2), (VP1,EP2).

Convém observar ainda que, de acordo com as regras para a determinação do tipo do resultado de uma operação binária, dadas em 3.1.2, um estado em que um dos operandos tenha tipo word não será estado inicial para nenhum nó final relativo a operação aritmética, lógica ou de comparação entre operandos byte. Poderá sê-lo apenas para operações de deslocamento entre operandos byte, se o primeiro acesso do estado referir-se a um operando byte. Da mesma forma, um estado em que ambos os operandos tenham tipo byte, nunca será estado inicial para nenhum nó final relativo a operação entre operandos word. As designações dos nós finais referentes a operações entre operandos byte terminam por "1", e entre operandos word por "2".

Estados: Além dos registradores A e HL são necessários pelo me-

nos dois outros registradores auxiliares para conter operandos byte, word ou endereços de operandos de ambos os tipos. Foram escolhidos para este fim os registradores B e DE. Os seguintes elementos devem portanto fazer parte do conjunto de acessos:

VB - valor do operando no registrador B
VDE - valor do operando no registrador DE
EDE1 - endereço do operando byte no registrador DE
EDE2 - endereço do operando word no registrador DE

O conjunto de acessos é, portanto:
A = AIM u AIP u AR, sendo:
AR ={VA, VB, VHL, VDE, EHL1, EHL2, EDE1, EDE2}.

O conjunto de estados, por sua vez, é:

E ={(x,y) AxA | x e y não se referem ao mesmo registrador} .

Estados Finais: Observando o conjunto de instruções do processador 8080, verificamos que para somar dois operandos byte, os seguintes estados precisam ser alcançados:

(VA,VB), (VB,VA), (EHL1,VA), (VA,EHL1), (VA,V1) ou (V1,VA).

Verificamos ainda que o mesmo se dá com relação às operações de "e", "ou" e "ou exclusivo" entre operandos byte. Além disso, o tamanho das instruções correspondentes a cada uma das operações é o mesmo para cada um dos estados, isto é, se escolhemos com função custo da aresta o tamanho da instrução que efetua a transformação entre os nós que a aresta une, as arestas

((VA,VB), ADI1), ((VA,VB),E1), ((VA,VB),OU1) e ((VA,VB),XOU1),

por exemplo, têm todas o mesmo custo. Daí, o caminho mínimo de qualquer nó até os nós ADI1, E1, OU1 e XOU1 é o mesmo, excetuando-se a última aresta percorrida.

Para permitir que tornemos explícita esta semelhança entre grupos de operações, definiremos como estado final de uma operação todo aquele que preceder imediatamente o nó final correspondente à operação, isto é:

(x,y) é estado final da operação correspondente ao nó z ∈ B se e somente se (x,y) R z.

Como, para as operações não providas pelo jogo de instruções do 8080, foram definidas rotinas intrínsecas que as implementassem, a forma como estas rotinas receberem seus operandos é que indicarão os estados finais associados a estas operações. Na implementação do compilador LPS/300, foi resolvido que cada rotina intrínseca com operandos byte teria dois pontos de entrada, a saber:

- primeiro operando no topo da pilha do processador, segundo operando no registrador A e

- primeiro operando no registrador A, segundo operando no registrador B.

Cada rotina intrínseca com operandos word, por sua vez, teria tres pontos de entrada, a saber:

- primeiro operando no topo da pilha do processador, segundo operando no registrador HL,

- primeiro operando no registrador DE, segundo operando no registrador HL e

- primeiro operando no registrador HL, segundo operando no registrador DE.

Operações Equivalentes: Definiremos como operações equivalentes aquelas que possuem exatamente os mesmos estados finais e que, para qualquer par de estados finais **a** e **b**, a diferença entre os

custos das arestas que unem os nós a e b aos nós correspondentes às operações é constante, isto é, sendo z e w os nós finais correspondentes às operações:

G(a,z) - G(a,w) = G(b,z) - G(b,w), para todos a e b estados finais de z e w.

Esta definição propicia uma grande economia do espaço ocupado pelos vetores de sucessores para os diversos nós finais, pois permite agrupar num mesmo vetor o conjunto de caminhos mínimos para todas as operações equivalentes a uma dada operação, a menos da última transformação, que pode ser guardada numa tabela á parte.
Além disso, dada esta tabela, uma classe de operações equivalentes só precisa ter um nó representante no digrafo do esquema para geração de operações binárias, e basta uma execução do algoritmo 1, com este nó como nó final, para determinar os caminhos mínimos para toda uma classe de operações equivalentes. O único cuidado necessário, além da definição desta tabela auxiliar, é emitir a instrução que ela indicar ao invés da instrução dada pelo rótulo f da última aresta do caminho mínimo. Note-se ainda que precisam ser encontrados os caminhos mínimos a partir de todos os nós do conjunto união de todos os conjuntos de nós iniciais do grupo de operações equivalentes.

Com relação ao esquema para gerações binárias do compilador LPS/300, temos os seguintes grupos de operações equivalentes e seus respectivos estados finais:

GRUPO1 = { ADI1,E1,OU1,XOU1}, com estados finais (VA,VB), (VB,VA), (EHL1,VA), (VA,EHL1), (VA,V1), (V1,VA);

GRUPO2 = {SUB1}, COM ESTADOS FINAIS (VA,VB), (VA,EHL1), (VA,V1) e (V1,V1);

GRUPO3 = {MUL1 , (comparações entre operandos byte)}, com estados finais (VA,VB), (VB,VA), (VP1,VA), (VA,VP1) e (V1,V1);

GRUPO4 ={DIV1,MOD1, (operações de deslocamento)}, com estados

finais (VA,VB), (VP1,VA) e (V1,V1);

GRUPO5 = { AD I2 } , com estados finais (VHL,DVE), (VDE,VHL), (V1,V2), (V2,V1) e (V2,V2);

GRUPO6 ={demais operações com operandos word}, com estados finais (VHL,VDE), (VDE,VHL), (VP2,VHL), (V1,V2), (V2,V1) e (V2,V2);

GRUPO7 = {ARM1} , com estados finais (E1,VA), (EHL1,VA) e (EDE1,VA); e

GRUPO8 ={ARM2}, com estados finais (E2,VHL), (EHL2,VDE).

Cabe observar que as comparações entre operandos byte estão todas no GRUPO3, que é o de operações comutativas com operandos byte realizadas através de chamada a rotina intrinseca, pois aquelas que não são originalmente comutativas (por exemplo, <), podem ser como tal consideradas se optarmos pela chamada à rotina reversa (no caso, >), quando os operandos estiverem na ordem contrária à estabelecida por um dos pontos de entrada. Para as operações word a comutatividade não foi considerada.

Definição do Esquema: Conforme já foi visto, o conjunto de acessos é

A = { RI1, RI2, E1, E2, V1, V2, VP1, VP2, EP1, EP2,

VA, VB, VHL, VDE, EHL1, EHL2, EDE1, EDE2},

e o conjunto de estados

E = { (x,y) ε A X A | x e y não se referem ao mesmo registrador }.

Valendo-nos da definição de operações equivalentes, podemos reduzir o conjunto de operações binárias intermediárias a um representante para cada grupo de operações equivalentes, ou seja

B = {ADI1,SUB1,MUL1,DIV1,ADI2,SUB2,ARM1,ARM2}.

Como o conjunto A tem 18 elementos e o conjunto de pares (x,y) que se referem ao mesmo registrador tem 1(registrador A) + 1(registrador B) + 3x3(registrador HL) + 3x3(registrador DE) = 20 elementos, o conjunto E tem 18x18 - 20 = 304 elementos, e o conjunto (E u B) tem 304 + 8 = 312 elementos. A definição das funções f e g se daria, portanto, sobre 312x312 = 100344 elementos.

Para simplificar esta tarefa, vamos nos valer de uma característica do processador 8080, comum a muitos processadores, nos quais a grande maioria das instruções modifica apenas o acesso a um operando do par. Na verdade, excetuando-se as operações binárias e as instruções XCHG e XTHL agindo sobre alguns estados, todas as outras instruções comportam-se desta maneira. Assim, começamos pela descrição do efeito das instruções sobre o acesso a um único operando, através das tabelas 6 e 7. Na tabela 6, o elemento da linha i, coluna j indica a instrução que transforma o acesso i no acesso j e na tabela 7 o mesmo elemento indica o tamanho desta instrução.

Alguns elementos da tabela 6 não se referem propriamente a instruções, mas a rotinas intrínsecas, tais como HLMM, que representa uma rotina que move um palavra de memória apontada por HL para o próprio HL, HLM, que represente uma rotina que mova o byte de memória apontado por HL para o próprio HL, expandindo o sinal, e HLA que faz o mesmo com o conteúdo do registrador A. As rotinas DEMM, DEM e DEA agem da mesma forma, mas o resultado vai para o registrador DE. O elemento EHLRI, por sua vez, indica uma sequência de instruções variável conforme o caso, seja a origem uma variável referida, indexada, duplamente indexada, redefinida e indexada, etc... Todavia, sendo uma transformação obrigatória sobre operandos RI1 ou RI2 (note-se que são os únicos elementos em suas respectivas linhas), seu custo pode ser arbitrariamente escolhido, e assim o foi o valor 5.

	EHL1	EHL2	EDE1	EDE2	VA	VB	VP1	VHL	VDE	VP2
RI1	EHLRI									
RI2		EHLRI								
E1					LDA					
E2								LHLD		
EHL1			XCHG		MOVAM	MOVBM		HLM	DEM	
EHL2				XCHG	MOVAM	MOVBM		HLMM	DEMM	
EDE1	XCHG				LDAXD					
EDE2		XCHG								
EP1	POPH		POPD							
EP2		POPH		POPD						
VA						MOVBA	PUSHA	HLA	DEA	
VB					MOVAB		PUSHB			
VP1					POPA	POPB				
VHL									XCHG	PUSHH
VDE								XCHG		PUSHD
VP2								POPH	POPD	
V1					MVIA	MVIB		LXIH	LXID	
V2					MVIA	MVIB		LXIH	LXID	

Tabela 6 - Transformações sobre um Operando

	EHL1	EHL2	EDE1	EDE2	VA	VB	VP1	VHL	VDE	VP2
RI1	5									
RI2		5								
E1					3					
E2								3		
EHL1			1		1	1		3	3	
EHL2				1	1	1		3	3	
EDE1	1				1					
EDE2		1								
EP1	1		1							
EP2		1		1						
VA						1	1	3	3	
VB					1		1			
VP1					1	1				
VHL									1	1
VDE								1		1
VP2								1	1	
V1					2	2		3	3	
V2					2	2		3	3	

Tabela 7 - Custos das Transformações sobre um Operando

A maior parte das arestas do esquema para geração de operações binárias do compilador LPS/300, bem como os valores de seus rótulos f e g, pode ser deduzida a partir das tabelas 8 e 9. Para tanto, basta criar uma aresta entre os nós (x,y) e (z,w), do conjunto E, sempre que estiver definido o elemento (x,z) da tabela 8 e y=w, ou se estiver definido o elemento (y,w) da tabela 8 e x=z. No primeiro caso teremos:

f((x,y), (z,w)) = elemento (x,z) da tabela 8;
g((x,y), (z,w)) = elemento (x,z) da tabela 9;

e no segundo caso:

f((x,y), (z,w)) = elemento (y,w) de tabela 8;
g((x,y), (z,w)) = elemento (y,w) da tabela 9.

Lembremos que os nós do dígrafo não são todos os elementos do conjunto AxA, mas apenas aqueles em que ambas as partes não se referem ao mesmo registrador.

Temos assim as arestas ((E2,E2),(VHL,E2)) e ((VA,VP1),(VA,VB))
por exemplo, sendo que

f((E2,E2),(VHL,E2)) = LHLD (elementos (E2,VHL) da tabela 6);
g((E2,E2),(VHL,E2)) = 3 (elemento (E2,VHL) da tabela 7);
f((VA,VP1),(VA,VB)) = POPB (elemento (VP1,VB) da tabela 6);
g((VA,VP1),(VA,VB)) = 1 (elemento (VP1,VB) da tabela 7).

As demais arestas e seus rótulos podem ser acrescentados diretamente, de acordo com as tabelas 8 e 9.

Nó. Origem	Nó. Destino	Instrução	Custo
(EHL1,EDE1)	(EDE1,EHL1)	XCHG	1
(EHL1,EDE2)	(EDE1,EHL2)	XCHG	1
(EHL1,VDE)	(EDE1,VHL)	XCHG	1
(EHL2,EDE1)	(EDE2,EHL1)	XCHG	1
(EHL2,EDE2)	(EDE2,EHL2)	XCHG	1
(EHL2,VDE)	(EDE2,VHL)	XCHG	1
(VHL,EDE1)	(VDE,EHL1)	XCHG	1
(VHL,EDE2)	(VDE,EHL2)	XCHG	1
(VHL,VDE)	(VDE,VHL)	XCHG	1
(EDE1,EHL1)	(EHL1,EDE1)	XCHG	1
(EDE1,EHL2)	(EHL1,EDE2)	XCHG	1
(EDE1,VHL)	(EHL1,VDE)	XCHG	1
(EDE2,EHL1)	(EHL2,EDE1)	XCHG	1
(EDE2,EHL2)	(EHL2,EDE2)	XCHG	1
(EDE2,VHL)	(EHL2,VDE)	XCHG	1
(VDE,EHL1)	(VHL,EDE1)	XCHG	1
(VDE,EHL2)	(VHL,EDE2)	XCHG	1
(VDE,VHL)	(VHL,VDE)	XCHG	1
(EHL1,EP1)	(EP1,EHL1)	XTHL	1
(EHL1,EP2)	(EP1,EHL2)	XTHL	1
(EHL1,VP2)	(EP1,VHL)	XTHL	1
(EHL2,EP1)	(EP2,EHL1)	XTHL	1
(EHL2,EP2)	(EH2,EHL2)	XTHL	1
(EHL2,VP2)	(EP2,VHL)	XTHL	1
(VHL,EP1)	(VP1,EHL1)	XTHL	1
(VHL,EP2)	(VP1,EHL2)	XTHL	1
(VHL,VP2)	(VP1,VHL)	XTHL	1
(EP1,EHL1)	(EHL1,EP1)	XTHL	1
(EP1,EHL2)	(EHL1,EP2)	XTHL	1
(EP1,VHL)	(EHL1,VP2)	XTHL	1
(EP2,EHL1)	(EHL2,EP1)	XTHL	1

Nó. Origem	Nó. Destino	Instrução	Custo
(EP2 ,EHL2)	(EHL2 ,EP2)	XTHL	1
(EP2 ,VHL)	(EHL2 ,VP2)	XTHL	1
(VP2 ,EHL1)	(VHL ,EP1)	XTHL	1
(VP2 ,EHL2)	(VHL ,EP2)	XTHL	1
(VP2 ,VHL)	(VHL ,VP2)	XTHL	1

Tabela 8 - Outras Arestas do Digrafo

Nó. Origem	Nó. Destino	Instrução	Custo
(VA,VB)	ADI1	ADDB	1
(VB,VA)	ADI1	ADDB	1
(VA,EHL1)	ADI1	ADDM	1
(EHL1,VA)	ADI1	ADDM	1
(VA,V1)	ADI1	ADI	2
(V1,VA)	ADI1	ADI	2
(V1,V1)	ADI1	---	0
(VA,VB)	MUL1	MULB	3
(VB,VA)	MUL1	MULB	3
(VA,VP1)	MUL1	MULP	3
(VP1,VA)	MUL1	MULP	3
(V1,V1)	MUL1	----	0
(VA,VB)	SUB1	SUBB	1
(VA,EHL1)	SUB1	SUBM	1
(VA,V1)	SUB1	SBI	2
(V1,V1)	SUB1	---	0
(VP1,VA)	DIV1	DIVP	3
(VA,VB)	DIV1	DIVB	3
(V1,V1)	DIV1	----	0
(VHL,VDE)	ADI2	DADD	1
(VDE,VHL)	ADI2	DADD	1
(V1,V2)	ADI2	----	0
(V2,V1)	ADI2	----	0
(V2,V2)	ADI2	----	0
(VP2,VHL)	SUB2	SUBPH	3
(VDE,VHL)	SUB2	SUBDH	3
(VHL,VDE)	SUB2	SUBHD	3
(V1,V2)	SUB2	-----	0
(V2,V1)	SUB2	-----	0
(V2,V2)	SUB2	-----	0
(E1,VA)	ARM1	STA	3

Nó. Origem	Nó. Destino	Instrução	Custo
(EDE1,VA)	ARM1	STAX	1
(EHL1,VA)	ARM1	MOVMA	1
(E2,VHL)	ARM2	SHLD	3
(EHL,VDE)	ARM2	MMDE	3

Tabela 9 - Arestas Finais

Também na tabela 9 temos alguns elementos da coluna "Instrução" que não representam instruções do processador e sim rotinas intrínsecas, tal como foi descrito no ítem "estados Finais". Estão neste caso:

MULB - refere-se ao segundo ponto de entrada da rotina de multiplicação de operandos byte: (VA,VB);

MULP - refere-se ao primeiro ponto de entrada desta rotina: (VP1,VA);

DIVB - refere-se ao segundo ponto de entrada da rotina de divisão de operandos byte: (VA,VB);

DIVP - refere-se ao primeiro ponto de entrada desta rotina: (VP1,VA);

SUBPH - refere-se ao primeiro ponto de entrada da rotina de subtração de operandos word: (VP2,VHL);

SUBDH - refere-se ao segundo ponto de entrada desta rotina: (VDE,VHL);

SUBHD - refere-se ao terceiro ponto de entrada desta rotina: (VHL,VDE);

MMDE - refere-se a uma rotina que move o conteúdo do registrador DE para a palavra de memória apontada por HL.

Caminhos Mínimos: Tendo sido definido acima o esquema para geração de operações binárias LPS/300, resta a aplicação do algoritmo 1 para a obtenção dos caminhos mínimos até os nós finais. Os resultados para o nó final SUB1 está na tabela 10, onde o elemento de linha i, coluna j, indica o sucessor do nó (i,j) no caminho mínimo até SUB1.

s.SUB1	RI1	E1	EHL1	VA	VB	V1
RI1	EHL1,RI1	EHL1,E1	RI1,VB	RI1,VB	EHL1,VB	EHL1,V1
E1	E1,EHL1	VA,E1	VA,EHL1	E1,VB	VA,VB	VA,V1
EHL1	VA,RI1	VA,E1		EHL1,VB	VA,VB	VA,V1
EDE1			VA,EHL1			
EP1			EDE1,EHL1	EP1,VB	EHL1,VB	
VA	VA,EHL1	VA,EHL1	SUB1		SUB1	SUB1
VP1			VA,EHL1	VP1,VB	VA,VB	
V1	V1,EHL1	V1,EHL1	VA,EHL1	V1,VB	VA,VB	SUB1

Tabela 10 - Caminhos Mínimos até SUB1

IV.3 - Esquema para Tradução de Operações Binárias no LPS/500

A aplicação do ETOB ao LPS/500 será vista brevemente, pois faz uso dos mesmos conceitos e simplificações adicionais já estudadas na seção precedente. O conjunto de instruções e os modos de endereçamento do processador do COBRA-500 são, no entanto, bastante diferentes dos já analisados para o COBRA-300. Assim, definiremos o ETOB para o COBRA-500, embora de forma mais direta, para proporcionar uma comparação entre as aplicações a dois processadores diferentes.

A pilha de ligação referida pelo código intermediário é uma área de memória apontada pelo registrador R1, crescendo dos endereços mais baixos para os mais altos. O registrador de estado corresponde ao bit Z da palavra de estado do processador, sendo o valor "verdadeiro" associado ao estado "ligado". A pilha de operandos será vista a seguir.

Pilha de Operandos: A pilha de operandos consta de um a seis registradores entre R2 e R7 inclusive, seguidos da pilha apontada por R1, também usada como pilha de ligação. A ordem dos registradores na pilha é qualquer, mas é alocado preferencialmente R2, depois R3 e assim por diante até R7. Este último é usado para retorno do resultado das funções. Na pilha de R1 todos os elementos ocupam uma palavra mesmo que sejam do tipo "byte".

Nós Finais: Assim como no LPS/300, cada operação binária intermediária dá origem a dois nós finais, um para o tipo "byte" e outro para "word". O dígito "1" após o nome da operação indicará tipo "byte" e o dígito "2" tipo "word".

Estados Iniciais: Os operandos endereçados indiretamente e com deslocamento têm seu endereço gerado previamente em registrador. Da mesma forma, os operandos indexados têm índice gerado previamente em registrador, bem como as constantes relocáveis têm seu

valor em registrador. Assim sendo, levando-se em conta a completa equivalência entre os registradores R2 até R7, são os seguintes os acessos iniciais, onde "registrador" refere-se um dentre estes:

M1 - operando byte em endereço de memória direta ou indiretamente
conhecido M2 - operando word idem
V1 - operando byte de valor conhecido
V2 - operando word idem
VR1 - operando byte com valor em registrador
VR2 - operando word idem
ER1 - operando byte com endereço em registrador
ER2 - operando word idem
IR1 - operando byte com endereço base direta ou indiretamente conhecido e índice em registrador
IR2 - operando word idem
VP1 - operando byte com valor no topo da pilha de R1
VP2 - operando word idem
EP1 - operando byte com endereço no topo da pilha de R1
EP2 - operando word idem
IP1 - operando byte com endereço direta ou indiretamente conhecido e índice no topo da pilha de R1
IP2 - operando word idem

Sendo AM = { M1, M2, V1, V2 }, AR = {VR1, VR2, ER1, ER2, IR1, IR2}, AP = {VP1, VP2, EP1, EP2, IP1, IP2} e A = AM u AR u AP, o conjunto de estados iniciais é

$EI = \{ (x,y) \in A \times (AM \cup AR) \mid x \in AP \Rightarrow y \in AR \}$.

Estados: Do conjunto de pares de acessos, eliminaremos aqueles em que o segundo elemento está na pilha de R1, pois não são nós iniciais e certamente não farão parte de nenhum caminho mínimo, além de introduzirem um caso não previsto no conjunto de acessos, com um operando no subtopo da pilha de R1. Assim
$E = A \times (AM \cup AR)$.

Estados FInais e Operações Equivalentes: São as seguintes as classes de operações equivalentes, com os respectivos estados finais:

GRUPO1 = { ADI1, (comparações aritméticas tipo byte) } , com estados finais (VR1,VR1), (VR1,ER1), (VR1,V1), (VR1,M1), (VR1,IR1) (V1,V1), (ER1,VR1), (VP1,VR1), (V1,VR1), (M1,VR1) e (IR1,VR1);

GRUPO2: { E1, OU1, XOU1, (comparações lógicas tipo byte) }, com os mesmos estados finais que o GRUPO1, mas constituem uma outra classe porque o custo das transformações finais para (VR1,V1) e (V1,VR1) é maior nesta classe;

GRUPO3: { operações de deslocamento tipo byte}, com estados finais (VR1,VR1), (VR1,V1), (VR1,ER1), (V1,V1), (VR1,VR2), (VR1,V2) e (V1,V2);

GRUPO4: {SUB1, DIV1, MOD1 }, com estados finais (VR1,VR1), (VR1,ER1), (V1,V1), (VR1,V1), (VR1,M1) e (VR1,IR1)

GRUPO5: { ARM1 }, com estados finais (ER1,VR1), (M1,VR1), (IR1,VR1), (M1,V1), (IR1,V1), (M1,ER1), (IR1,ER1), (ER1,M1), (ER1,IR1), (ER1,VR2), (M1,VR2), (M1,V2), (IR1,V2);

GRUPO6: { ADI2, E2, OU2, XOU2, (comparações word)}, com estados finais (VR2,VR2), (VR2,ER2), (VR2,V2), (VR2,M2), (VR2,IR2), (V1,V2), (V2,V1), (V2,V2), (ER2,VR2), (V2,VR2), (M2,VR2), (IR2,VR2) e (VP2,VR2);

GRUPO7: {operações de deslocamento tipo word}, com estados finais
(VR2,VR2), (VR2,V2), (VR2,ER2), (V2,V2), (VR2,VR1), (VR2,V1), (V2,V1);

GRUPO8: { SUB2, DIV2, MOD2}, com estados finais
(VR2,VR2), (VR2,ER2), (VR2,V2), (VR2,M2), (VR2,IR2), (V1,V2), (V2,V1) e (V2,V2);

GRUPO9: { ARM2 }, com estados finais (ER2 ,VR2), (M2 ,VR2), (IR2 ,VR2), (M2 ,V2), (IR2 ,V2), (M2 ,ER2), (IR2 ,ER2), (ER2 ,M2), e (ER2 ,IR2).

Definição do Esquema: As tabelas 12 e 13 contém as transformações sobre um operando e seu custo, respectivamente. Como todas as instruções utilizadas exceto as operações binárias agem sobre um único operando, podemos criar no digrafo do ETOB arestas entre os nós (x,y) e (z,w) sempre que estiver definido o elemento (x,z) da tabela 12 e y = w, ou se estiver definido (y,w) e x = z. No primeiro caso
f((x,y),(z,w)) = elemento (x,z) da tabela 12;
g((x,y),(z,w)) = elemento (x,z) da tabela 13; e no segundo caso
f((x,y),(z,w)) = elemento (y,w) da tabela 12;
g((x,y),(z,w)) = elemento (y,w) da tabela 13.

	VR1	VR2	ER1	ER2	IR1	IR2	VP1	VP2	EP1	EP2	IP1	IP2
M1	CGB	CGB	CGE						CGE			
M2		CG		CGE				CG		CGE		
V1	CGIB	CGIB					CGE	CGE				
V2	CGIB	CGE					CGE	CGE				
VR1		EBD					AR					
VR2	--						AR	AR				
ER1	CGRB	CGRB							AR			
ER2	CGR	CGR	INC				CG	CG		AR		
IR1	CGB	CGB	CGE						CGE		AR	
IR2		CG		CGE	INC			CG		CGE		AR
VP1	CGR							EBD				
VP2	CGR	CGE					--					
EP1			CGR									
EP2				CGR						INC		
IP1					AR							
IP2						AR					INC	

Tabela 12 - Transformações sobre um Operando

	VR1	VR2	ER1	ER2	IR1	IR2	VP1	VP2	EP1	EP2	IP1	IP2
M1	2	2	2						2			
M2		2		2				2		2		
V1	1	1					2	2				
V2	1	2					2	2				
VR1		1					1					
VR2	0						1	1				
ER1	1	1							1			
ER2	1	1	1				2	2		1		
IR1	2	2	2						2		1	
IR2		2		2	1			2		2		1
VP1	1							1				
VP2	1	1					0					
EP1			1									
EP2				1					1			
IP1					1							
IP2						1					1	

Tabela 13 - Custo das Transformações Sobre um Operando

Para completar o esquema faltam as arestas finais, isto é, aquelas cujo nó destino é um dos representantes das classes de operações equivalentes. Na tabela 14 apresentamos as arestas finais, juntamente com seus rótulos, para o nó final ADI1. Nesta, a letra "X" indica o primeiro operando e a letra "Y" o segundo, "R" indica registro de R2 a R7, "V" um valor constante e "M" um endereço de memória direto ou indireto.

Nó Origem	Nó Destino	Instrução	Custo
(VR1,VR1)	ADI1	ADR RX,RY	1
(VR1,V1)	ADI1	ADI RX,VY	1
(VR1,ER1)	ADI1	ADR RX,(RY)	1
(VR1,M1)	ADI1	AD RX,MY	2
(VR1,IR1)	ADI1	AD RX,MY(RY)	2
(V1,V1)	ADI1	---	0
(ER1,VR1)	ADI1	ADR RY,(RX)	1
(VP1,VR1)	ADI1	ADR RY,(-R1)	1
(V1,VR1)	ADI1	ADI RY,VX	1
(M1,VP1)	ADI1	AD RY,MX	2
(IR1,VR1)	ADI1	AD RY,MX(RX)	2

Tabela 14 - Aresta Finais para ADI1

Caminhos Mínimos: Definido o esquema, a aplicação do algoritmo 1 fornece os caminhos mínimos até os nós AD I1, onde o elemento da linha i, coluna j, indica o sucessor do nó (i,j) no caminho até AD I1.

s.AD I1	M1	V1	VR1	ER1	IR1
M1	M1,VR1	VR1,V1	VR I1	VR1,ER1	M1,VR1
V1	V1,VR1	AD I1	AD I1	V1,VR1	V1,VR1
VR1	AD I1	AD I1	AD I1	AD I1	AD I1
ER1	ER1,VR1	VR1,V1	AD I1	VR1,ER1	ER1,VR1
IR1	VR1,M1	VR1,V1	AD I1	VR1,ER1	VR1,IR1
VP1	VR1,M1	AD I1	VR1,ER1	VR1,ER1	VR1,IR1
EP1	EP1,VR1	ER1,V1	ER1,VR1	ER1,ER1	EP1,VR1
IP1	IR1,M1	IR1,V1	IR1,IR1	IR1,ER1	IR1,IR1

Tabela 15 - Caminhos Mínimos até AD I1

V - Conclusão

A geração de código por um compilador pode ser dividida em três fases: tradução do programa fonte numa sequência de chamadas a rotinas semânticas; execução das rotinas semânticas, quando é feita a geração do código intermediário e o cálculo de atributos semânticos; e tradução das instruções intermediárias em instruções do processador. As gramáticas de tradução livres-de-contexto descrevem a primeira fase, possibilitando sua implementação automática ou pelo menos sistemática. O processo de tradução, efetuado em conjunto com análise sintática ascendente, pode consistir simplesmente na emissão dos símbolos de saída de cada produção usada na análise sintática, desde que na gramática de tradução não ocorram terminais de saída à esquerda de não-terminais em nenhuma produção. A gramática original deve ser transformada para atender a esta condição. Se a análise sintática se faz por matriz de transição, a própria transformação da gramática de operadores em gramática aumentada pode ser aproveitada, mas a análise sintática deverá ser feita sobre esta última. O algoritmo de obtenção da gramática aumentada pode ser facilmente estendido a gramáticas de tradução livres-de-contexto de operadores nas quais cadeias de terminais de saída figurem sempre após um terminal de entrada ou no final da produção.

Quanto à segunda fase da geração de código, as gramáticas de tradução-livre-de-contexto com atributos proporcionam um meio de descrição da propagação dos atributos semânticos, mas a especificação do código intermediário gerado por cada rotina semântica e dos atributos produzidos é feita em forma de texto de programa, ou em linguagem corrente. Cada rotina semântica, em análise sintática ascendente, tem acesso aos atributos criados pelas rotinas que a antecedem no percorrimento da árvore de derivação na GTLCA em pré-ordem.

Quando a compilação é feita em um passo, ou mais exatamente quando cada instrução intermediária emitida é imediatamente traduzida em instruções do processador, a terceira fase da geração de código só apresenta dificuldades com relação às operações binárias. O problema de tradução de uma binária intermediária em

instruções do processador, porém, resume-se à determinação do caminho de custo mínimo entre o nó representante da localização do par de operandos em têrmos de acesso pelo processador e o nó representante da operação binária intermediária, num digrafo com nós do tipo do primeiro para todas as situações em que possa se encontrar um par de operandos e do tipo do segundo para as operações binárias intermediárias, e cada aresta rotulada duplamente com uma instrução do processador e um custo atribuído à instrução. Denominamos um tal digrafo duplamente rotulado de esquema de tradução de operações binárias (ETOB).

Como este problema de caminho mínimo pode ser resolvido algoritmicamente, e apenas esta última fase da geração de código é dependente do processador objeto, a portabilidade de um compilador escrito segundo este método é obtida pela simples redefinição do ETOB. A aplicação de uma variante do algoritmo de Dijkstra para determinação do caminho de custo mínimo entre dois nós, usando como partida o nó representante da operação binária, e como objetivo o conjunto de nós representantes das situações em que podem se encontrar os operandos da instrução intermediária, permite a determinação antecipada de todos os caminhos mínimos de interesse, que podem ser armazenados em tabelas no compilador. As operações binárias intermediárias podem ser agrupadas em classes de operações efetuáveis sobre pares de operandos na mesma situação quanto ao acesso pelo processador, e só precisa haver uma tabela de sucessores dos nós para cada classe.

O método foi empregado na implementação dos compiladores LPS para os computadores da linha COBRA-300 e COBRA-500, os quais possuem arquitetura diversas, o primeiro utilizando um micro-processador INTEL-8080 e o segundo uma UCP própria, projetada e construída na COBRA. Nesta aplicação, uma das metas foi produzir-se computadores semelhantes ao máximo para os dois processadores, no que o método descrito foi de grande utilidade. Na linguagem LPS está escrito todo o software dos produtos da linha COBRA-300 e boa parte do software dos produtos COBRA-500, sendo a LPS hoje uma ferramenta fundamental no desenvolvimento de programas facilmente portáteis entre os produtos COBRA.

Apêndice: Atributos Semânticos para LPS

1 - Atributo DADO

Descreve o localização dinâmica e o tipo de um dado.
É composto de um descritor da base seguido de zero ou mais descritores de índices. Cada descritor, seja de base ou de índice, é formado pelos seguintes campos:

LOC - é o valor do dado, seu endereço em memória ou uma indicação (%) de que o dado se encontra na pilha de operandos.

REL - caso o dado não esteja na pilha de operandos, fornece a relocação a que deve ser submetido o valor em LOC. Pode assumir os valores:

- "abs" - relocação absoluta
- "rd" - relocação em dados
- "ext" - referência externa

TIPO - indica o tipo do dado, podendo valer "byte" ou "word".

INDIR - pode valer:

- "valor" - LOC é o valor do dado
- "ender" - LOC é o endereço do dado na memória
- "ref" - LOC fornece o endereço de uma palavra de memória que contém o endereço do dado na memória

DESLOC - caso INDIR seja igual a "ref", fornece um valor a ser somado ao conteúdo da posição de memória dada por LOC para se obter o endereço do dado.

ÍNDICE - indica se este é um descritor de base ou de índice. No primeiro caso assume o valor "não" e no segundo o valor "sim"

2 - Atributo ROT

Guarda as informações necessárias sobre um procedimento, durante

a compilação da chamada. É formado pelos seguintes campos:

LOC - endereço de execução do procedimento

REL - relocação do valor em LOC. Pode assumir os valores:
"abs" - relocação absoluta
"rp" - relocação em programa
"ext" - referência externa

TIPO - indica o tipo da função, "byte" ou "word", ou que se trata de um subrotina, quando assume o valor "proc"

PARMATUAL - indica que parâmetro está sendo compilado correntemente

NPARMS - fornece o número total de parâmetros

TIPOPARM - é uma lista dos tipos dos parâmetros, "byte" ou "word"

3 - Atributo DROT

Guarda as informações necessárias sobre um procedimento, durante a compilação de sua declaração. É formado pelos seguintes campos:

LOC - endereço inicial da área de dados do procedimento

TIPO - vide atributo ROT

NPARMS - vide atributo ROT

TIPOPARM - vide atributo ROT

4 - Atributo PROG

Guarda um endereço de programa, definido ou não. Compõe-se de dois campos:

LOC - valor do endereço ou número da referência à frente

REL - indica a relocação do endereço, ou que se trata de número de referência à frente. Pode assumir os valores:

"rp" - relocação em programa
"indef" - número de referência à frente

5 - Atributo TIPO

Guarda o tipo de uma declaração, podendo assumir os valores "byte" e "word".

Bibliografia:

1. Aho, Alfred V., Ullman, Jeffrey D. - The Theory of Parsing, Translation and Compiling, Volume 1, Prentice-Hall, 1972.

2. Aho, Alfred V., Ullman, Jeffrey D. - Principles of Compiler Design, Addison-Wesley, 1976.

3. Bauer, F.L., Eickel, J. - Compiler Construction: An Advanced Course, Springer-Verlag, 1976.

4. Berztiss, A.T. - Data Structures: Theory and Practice, Academic Press, 1975.

5. Deo, Narsing - Graph Theory with Applications to Engineering and Computer Science, Pretice-Hall, 1974.

6. Gries, D. - Compiler Construction for Digital Computers, John Wiley & Sons, 1971.

7. Lewi, J., de Vlaminck, K., Huens, J., Huybrechts, M. - A Programming Methodology in Compiler Construction, North-Holland, 1979.

8. Lewis, P.M., Rosenkrantz, D.J., Stearns, R.E. Compiler Design Theory, Addison-Wesley, 1976.

9. MacLane, S., Birkhoff, G. - Algebra, MacMillan, 1967.

10. Manual de Referência da Linguagem LPS- COBRA-300, COBRA S/A, 1980

11. Manual de Referência da Linguagem LPS- COBRA-500, COBRA S/A, 1980.